Lecture 4: k-layer Neural Networks & Training neural networks

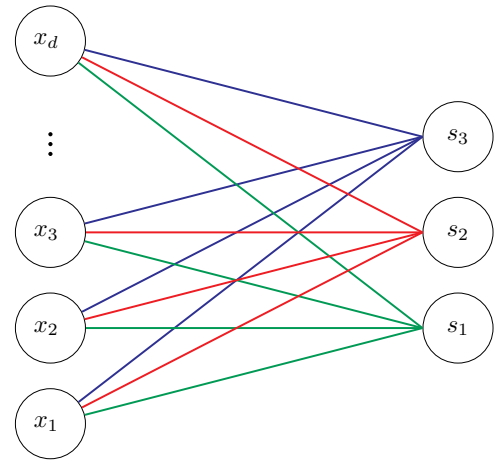
DD2424, Josephine Sullivan

March 25, 2025

A new class of scoring functions

Linear scoring function

$$\mathbf{s} = W\mathbf{x} + \mathbf{b}$$

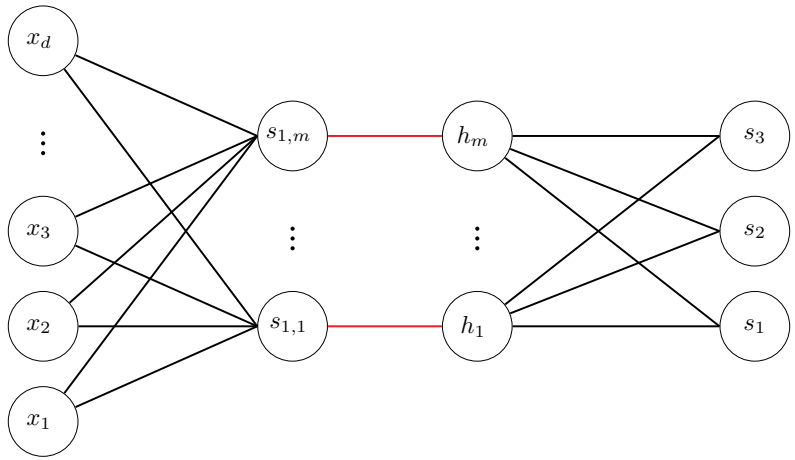


Input: $\mathbf{x}$ Output: $\mathbf{s} = W\mathbf{x} + \mathbf{b}$

Before

2-layer Neural Network

$$\mathbf{s}_1 = W_1\mathbf{x} + \mathbf{b}_1$$
$$\mathbf{h} = \max(0, \mathbf{s}_1)$$
$$\mathbf{s} = W_2\mathbf{h} + \mathbf{b}_2$$



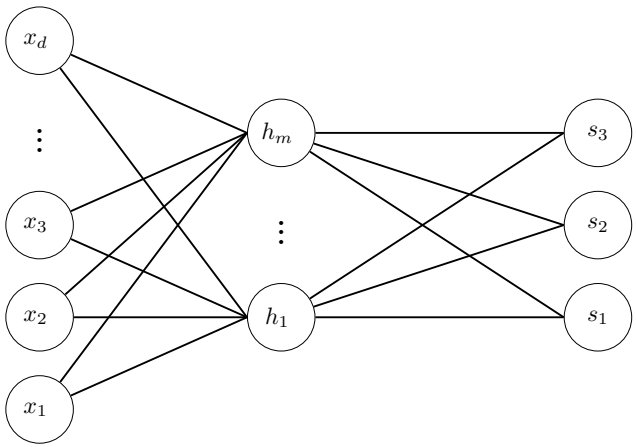
Input: $\mathbf{x}$ $\mathbf{s}_1 = W_1\mathbf{x} + \mathbf{b}_1$ $\mathbf{h} = \max(0, \mathbf{s}_1)$ $\mathbf{s} = W_2\mathbf{h} + \mathbf{b}_2$

Now

Not restricted to two layers

2-layer Neural Network

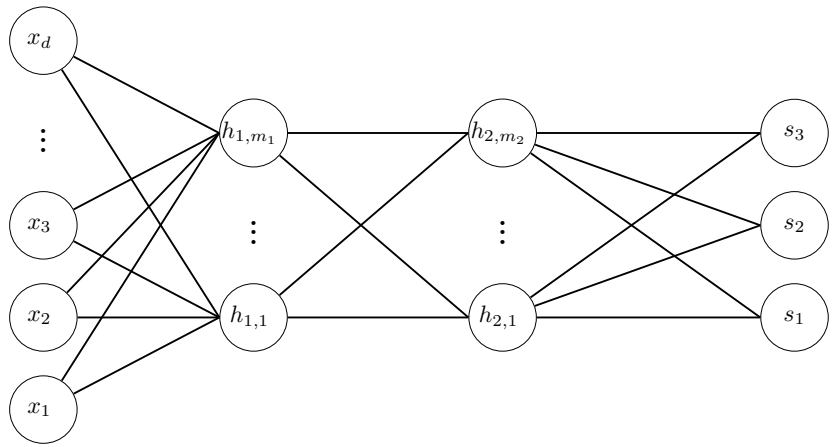
$$\mathbf{s}_1 = W_1\mathbf{x} + \mathbf{b}_1$$
$$\mathbf{h} = \max(0, \mathbf{s}_1)$$
$$\mathbf{s} = W_2\mathbf{h} + \mathbf{b}_2$$



Input: $\mathbf{x}$ $\quad$ $\mathbf{s}_1 = W_1\mathbf{x} + \mathbf{b}_1$ $\quad$ $\mathbf{h} = \max(0, \mathbf{s}_1)$ $\quad$ Output: $\mathbf{s} = W_2\mathbf{h} + \mathbf{b}_2$

3-layer Neural Network

$$\mathbf{s}_1 = W_1\mathbf{x} + \mathbf{b}_1$$
$$\mathbf{h}_1 = \max(0, \mathbf{s}_1)$$
$$\mathbf{s}_2 = W_2\mathbf{h}_1 + \mathbf{b}_2$$
$$\mathbf{h}_2 = \max(0, \mathbf{s}_2)$$
$$\mathbf{s} = W_3\mathbf{h}_2 + \mathbf{b}_3$$



Input: $\mathbf{x}$ $\quad$ $\mathbf{s}_1 = W_1\mathbf{x} + \mathbf{b}_1$ $\quad$ $\mathbf{h}_1 = \max(0, \mathbf{s}_1)$ $\quad$ $\mathbf{s}_2 = W_2\mathbf{h}_1 + \mathbf{b}_2$ $\quad$ $\mathbf{h}_2 = \max(0, \mathbf{s}_2)$ $\quad$ Output: $\mathbf{s} = W_3\mathbf{h}_2 + \mathbf{b}_3$

Some terminology

3-layer Neural Network

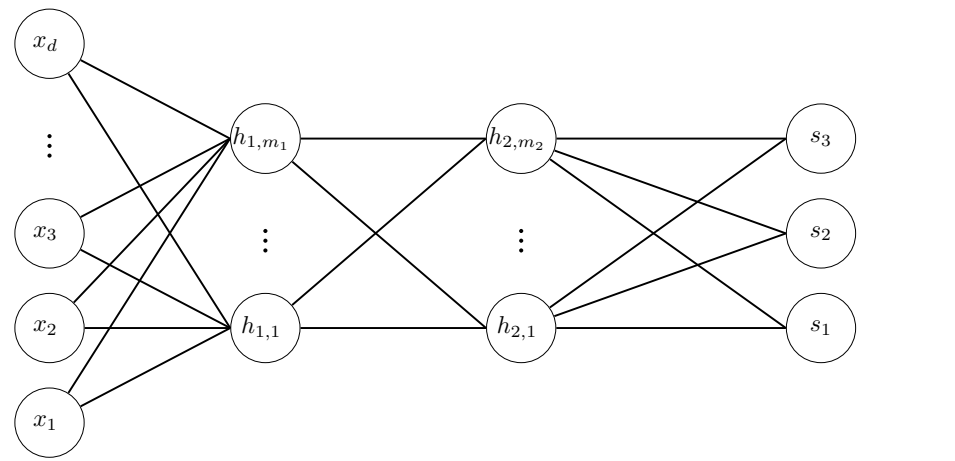
$$\mathbf{s}_1 = W_1\mathbf{x} + \mathbf{b}_1 \quad W_1 \text{ is } m_1 \times d$$

1st hidden layer activations $\rightarrow \mathbf{h}_1 = \max(0, \mathbf{s}_1)$ $\leftarrow$ apply non-linearity via **activation fn**

$$\mathbf{s}_2 = W_2\mathbf{h}_1 + \mathbf{b}_2 \quad W_2 \text{ is } m_2 \times m_1$$

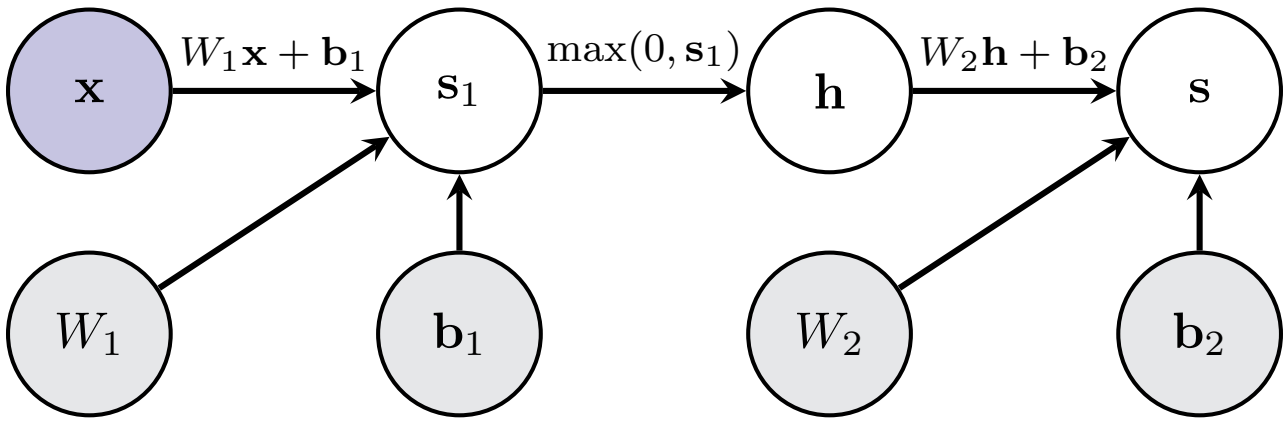
2nd hidden layer activations $\rightarrow \mathbf{h}_2 = \max(0, \mathbf{s}_2)$ $\leftarrow$ apply non-linearity via **activation fn**

Output responses $\rightarrow \mathbf{s} = W_3\mathbf{h}_2 + \mathbf{b}_3 \quad W_3 \text{ is } C \times m_2$

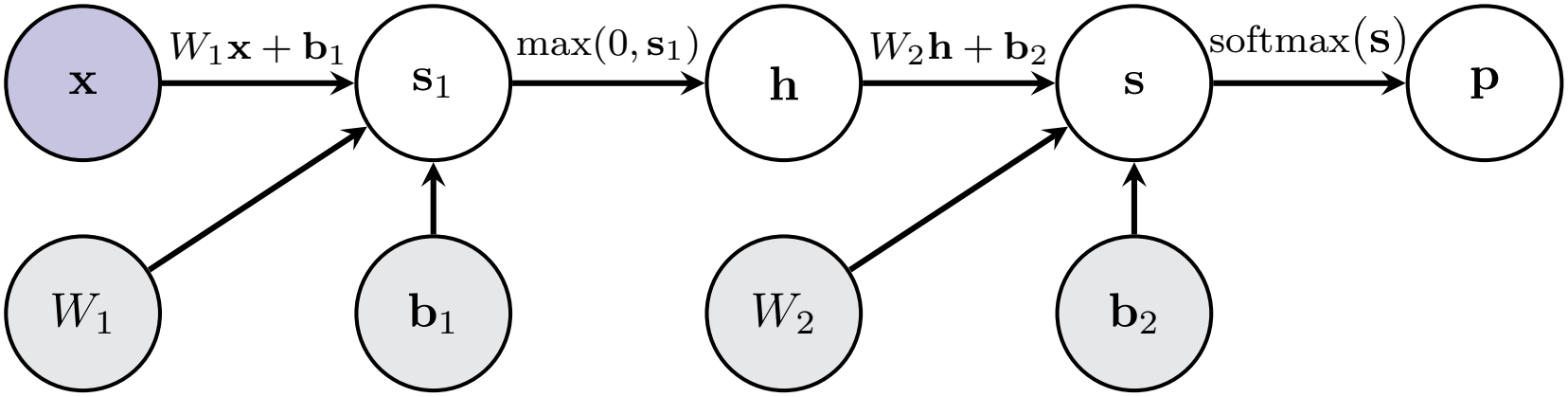


Sometimes referred to as a **2-hidden-layer neural network**.

Computational Graph of our 2-layer neural network

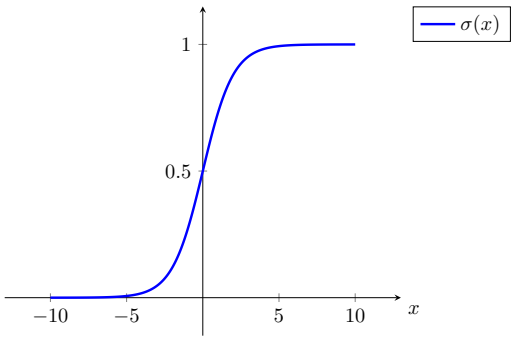


2-layer neural network with probabilistic outputs



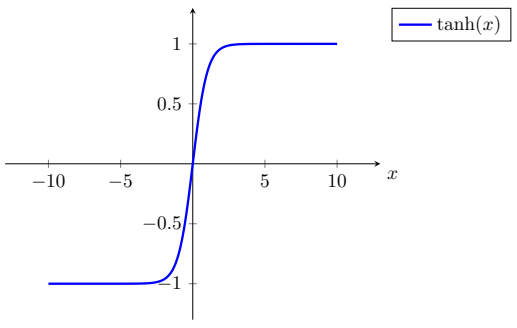
Options for Activation Functions

Sigmoid



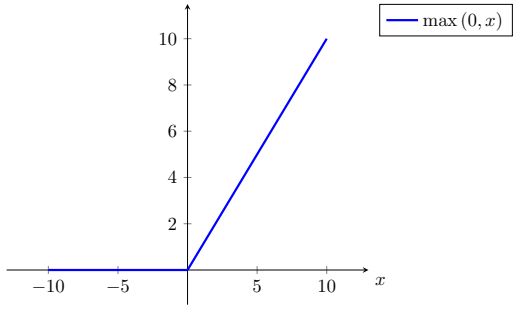
$$\sigma(x) = \frac{1}{1+\exp(-x)}$$

tanh



$$\tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$$

ReLu

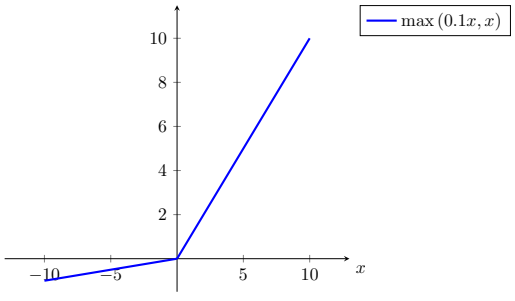


$$\text{ReLu}(x) = \max(0, x)$$

Activation function is applied independently to each element of hidden vector.

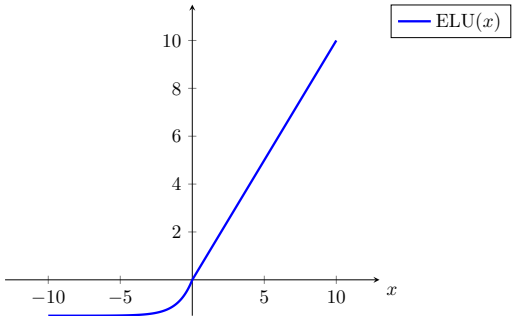
Options for Activation Functions

Leaky ReLu



$$\max(0.1x, x)$$

ELU

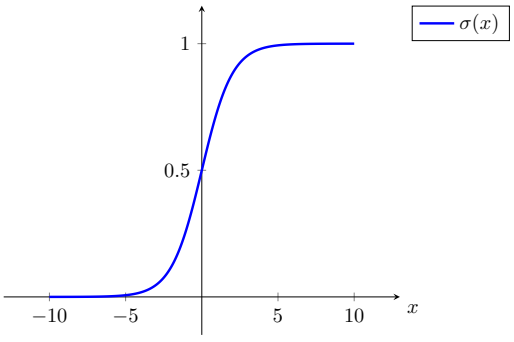


$$\text{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\left(\exp(x) - 1\right) & \text{otherwise} \end{cases}$$

Activation function is applied independently to each element of hidden vector.

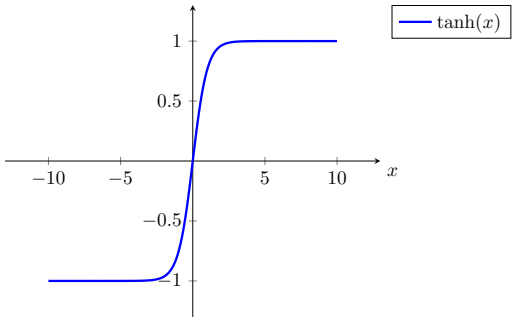
Options for Activation Functions

Sigmoid



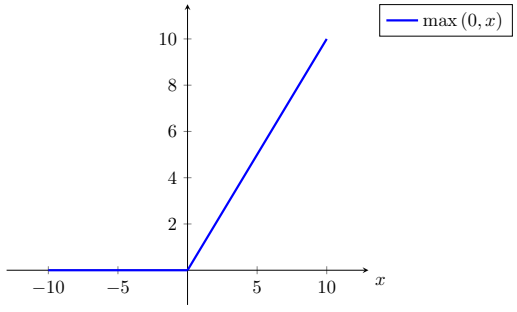
$$\sigma(x) = \frac{1}{1+\exp(-x)}$$

tanh



$$\tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$$

ReLu



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

✓

In modern networks ReLU is the most common activation function.

Many variations have been explored

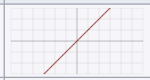
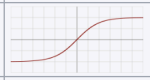
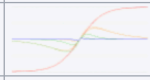
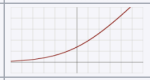
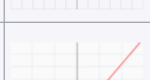
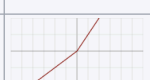
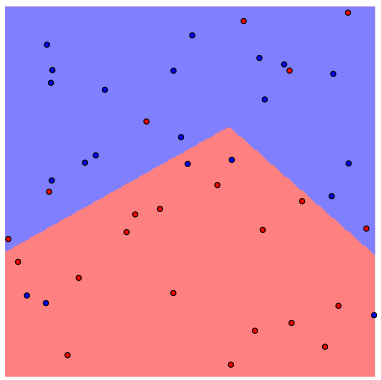
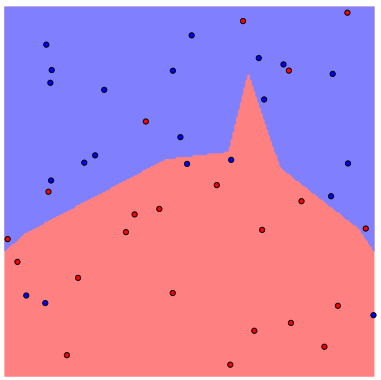
Name	Plot	Function, $g(x)$	Derivative of g, $g'(x)$	Range	Order of continuity
Identity		x	1	$(-\infty, \infty)$	C^∞
Binary step		$\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	0	$\{0, 1\}$	C^{-1}
Logistic, sigmoid, or soft step		$\sigma(x) \doteq \frac{1}{1+e^{-x}}$	$g(x)(1-g(x))$	$(0, 1)$	C^∞
Hyperbolic tangent (tanh)		$\tanh(x) \doteq \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$1 - g(x)^2$	$(-1, 1)$	C^∞
Soboleva modified hyperbolic tangent (smht)		$\operatorname{smht}(x) \doteq \frac{e^{ax} - e^{-bx}}{e^{cx} + e^{-dx}}$		$(-1, 1)$	C^∞
Rectified linear unit (ReLU) [8]		$(x)^+ \doteq \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$ $= \max(0, x) = x\mathbf{1}_{x>0}$	$\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$	$[0, \infty)$	C^0
Gaussian Error Linear Unit (GELU)[2]		$\frac{1}{2}x\left(1 + \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$ $= x\Phi(x)$	$\Phi(x) + x\phi(x)$	$(-0.17\ldots, \infty)$	C^∞
Softplus[9]		$\ln(1 + e^x)$	$\frac{1}{1+e^{-x}}$	$(0, \infty)$	C^∞
Exponential linear unit (ELU)[10]		$\begin{cases} \alpha\left(e^x - 1\right) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$ with parameter α	$\begin{cases} \alpha e^x & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$	$(-\alpha, \infty)$	$\begin{cases} C^1 & \text{if } \alpha = 1 \\ C^0 & \text{otherwise} \end{cases}$
Scaled exponential linear unit (SELU)[11]		$\lambda\begin{cases} \alpha(e^x - 1) & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$ with parameters $\lambda = 1.0507$ and $\alpha = 1.67326$	$\lambda\begin{cases} \alpha e^x & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$(-\lambda\alpha, \infty)$	C^0
Leaky rectified linear unit (Leaky ReLU)[12]		$\begin{cases} 0.01x & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$	$\begin{cases} 0.01 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$	$(-\infty, \infty)$	C^0
Parametric rectified linear unit (PReLU)[13]		$\begin{cases} \alpha x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$ with parameter α	$\begin{cases} \alpha & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$(-\infty, \infty)$	C^0
Sigmoid linear unit (SiLU,[2] Sigmoid shrinkage,[14] SiL,[15] or Swish-1[16])		$\frac{x}{1+e^{-x}}$	$\frac{1 + e^{-x} + xe^{-x}}{(1+e^{-x})^2}$	$[-0.278\ldots, \infty)$	C^∞

Image Source: Wikipedia page - Activation function

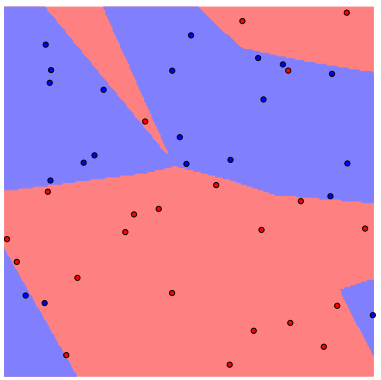
Effect of the number of hidden nodes in a 2 layer network



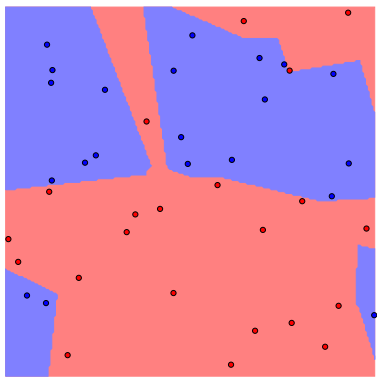
$m = 3$



$m = 20$



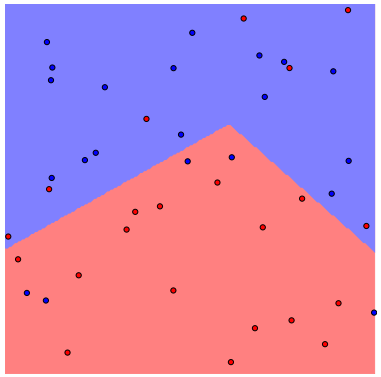
$m = 30$



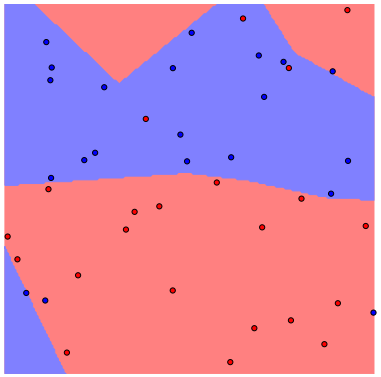
$m = 100$

- m is the number of nodes in the hidden layer.
- No regularization.

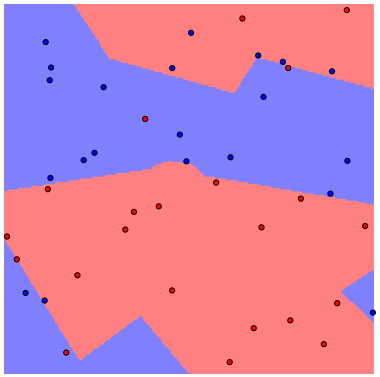
Result depends on parameter initialization



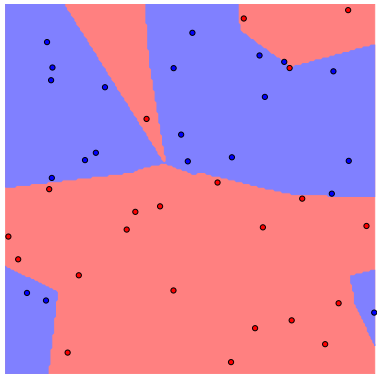
$m = 3$



$m = 20$



$m = 30$

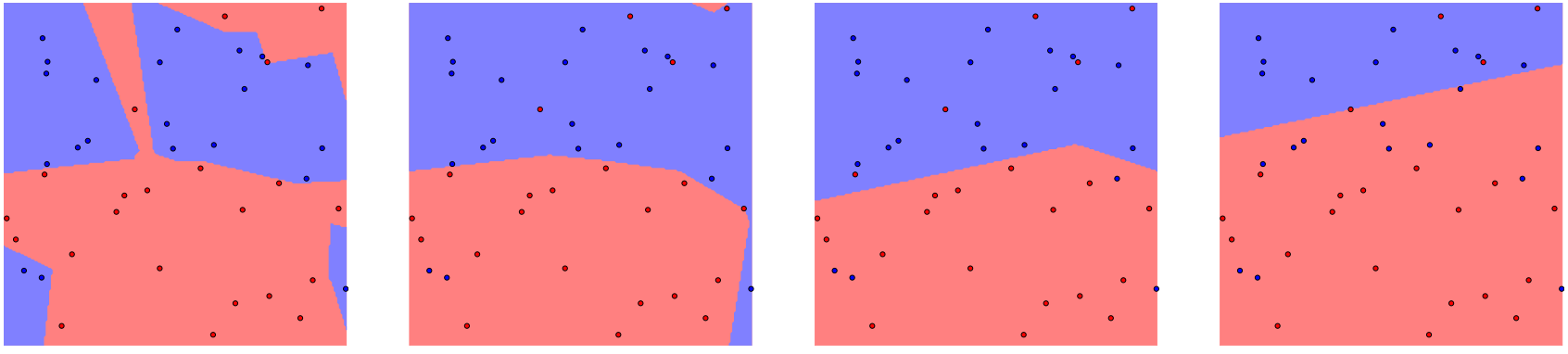


$m = 100$

- m is the number of nodes in the hidden layer.
- No regularization.
- Different random parameter initialization to previous slide.

Effect of regularization

$$J(\mathcal{D}, \lambda, \Theta) = \sum_{(\mathbf{x},y)\in\mathcal{D}} l(\mathbf{x}, y, \Theta) + \lambda R(\Theta)$$



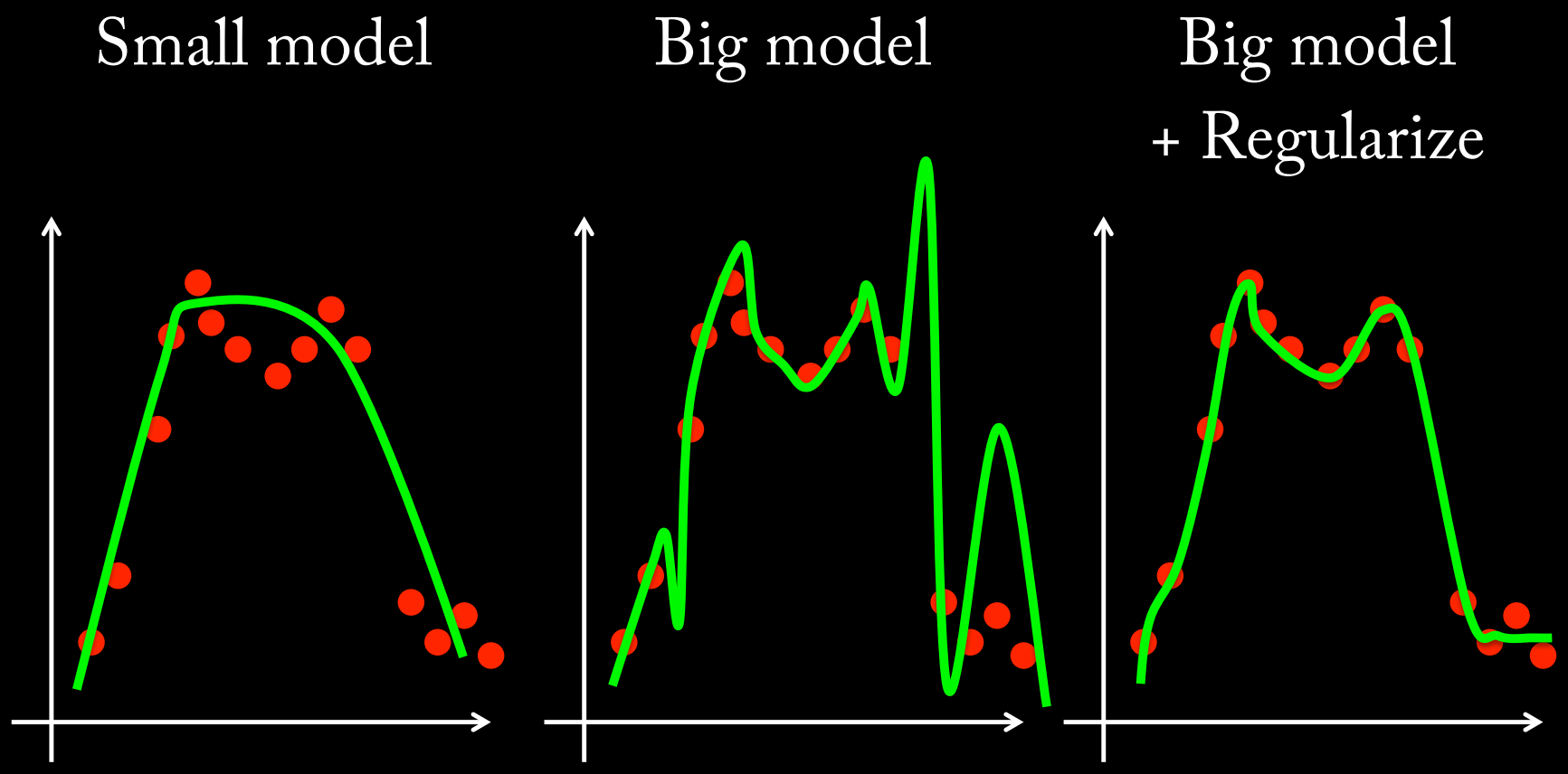
$\lambda = 0$ $\lambda = .001$ $\lambda = .01$ $\lambda = .1$

- $m = 100$ nodes in the hidden layer.
- L_2 regularization.

Do not use size of neural network as a regularizer.

Use stronger regularization.

Big Model + Regularize vs Small Model



Slight digression: Generalization & overfitting not so well understood for neural networks

The Mystery of Generalization in Deep Learning

- GoogLeNet, VGGNet,.... have many millions of parameters.
- Networks trained with (ignoring data augmentations)
  # training points $\ll$ # of parameters.
- But these networks still generalize well.
- What's going on??

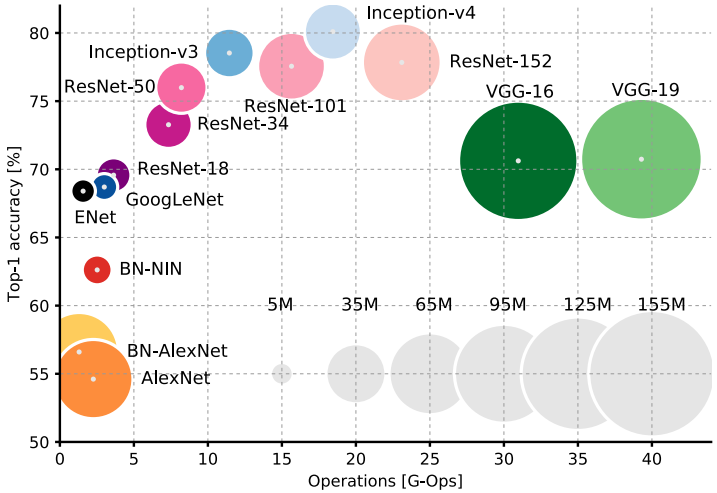


Fig credit: **An Analysis of Deep Neural Network Models for Practical Applications** by Canziani, Culurciello & Paszke.

End of digression

High-level overview of how to train network

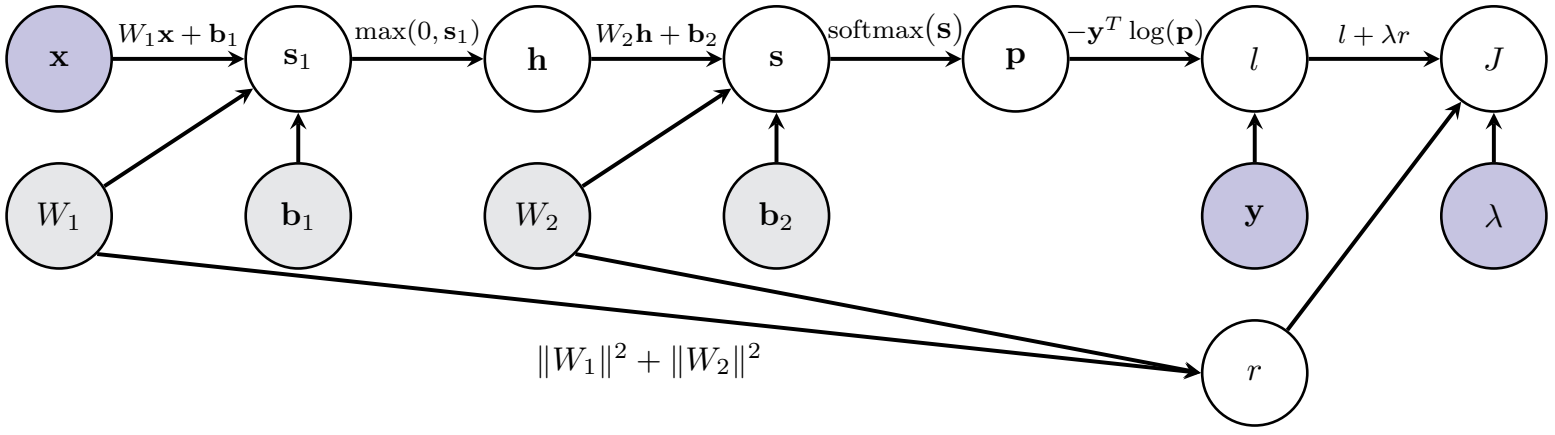
Mini-batch SGD (or variant)

Loop

1. **Sample** a batch of the training data.
2. **Forward propagate** it through the graph and calculate loss/cost.
3. **Backward propagate** to calculate the gradients.
4. **Update** the parameters using the gradient.

Gradient Computations for a k-layer neural network

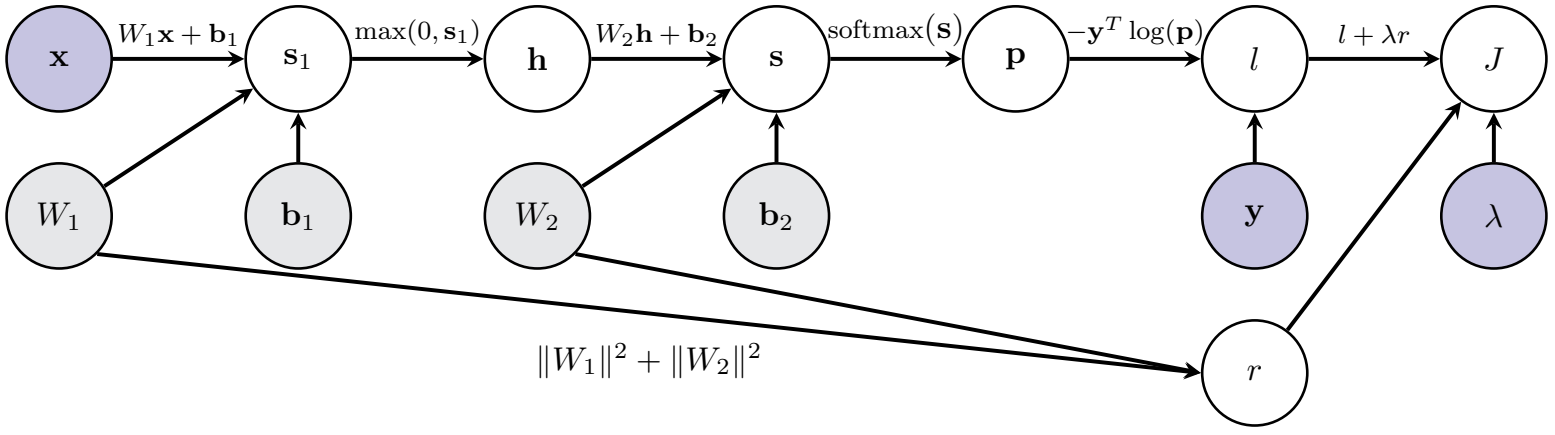
Back propagation for 2-layer neural network



For a single labelled training example:

1. **Forward propagate** it through the graph and calculate loss.
2. **Backward propagate** to calculate the gradients.

Back propagation for 2-layer neural network



For a single labelled training example:

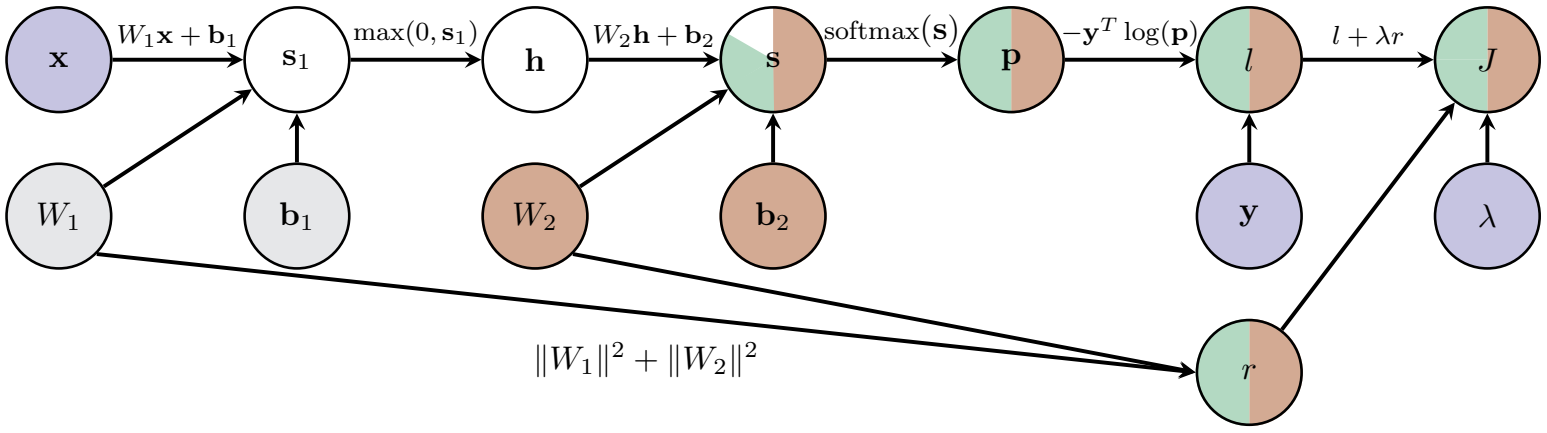
1. **Forward propagate** it through the graph and calculate loss.

   ↑ this is straightforward.

2. **Backward propagate** to calculate the gradients. ← Focus on this.

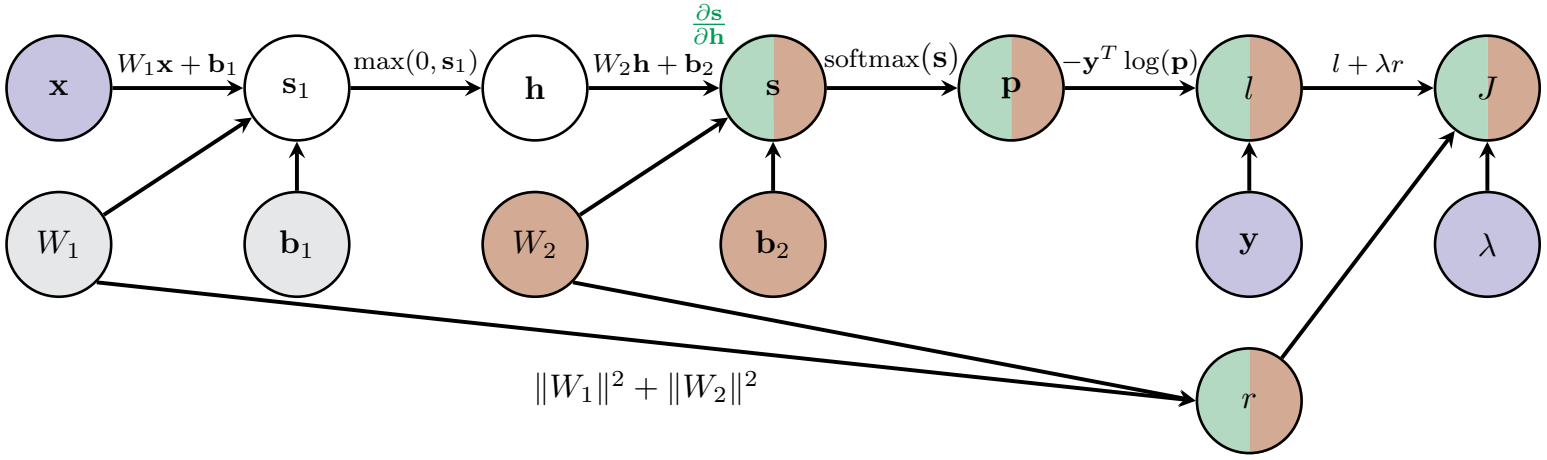
Backward Pass: Gradient of current node

Starting point of our demonstration



In Lecture 3 explicitly computed **filled in** *local Jacobians* and *gradients*.

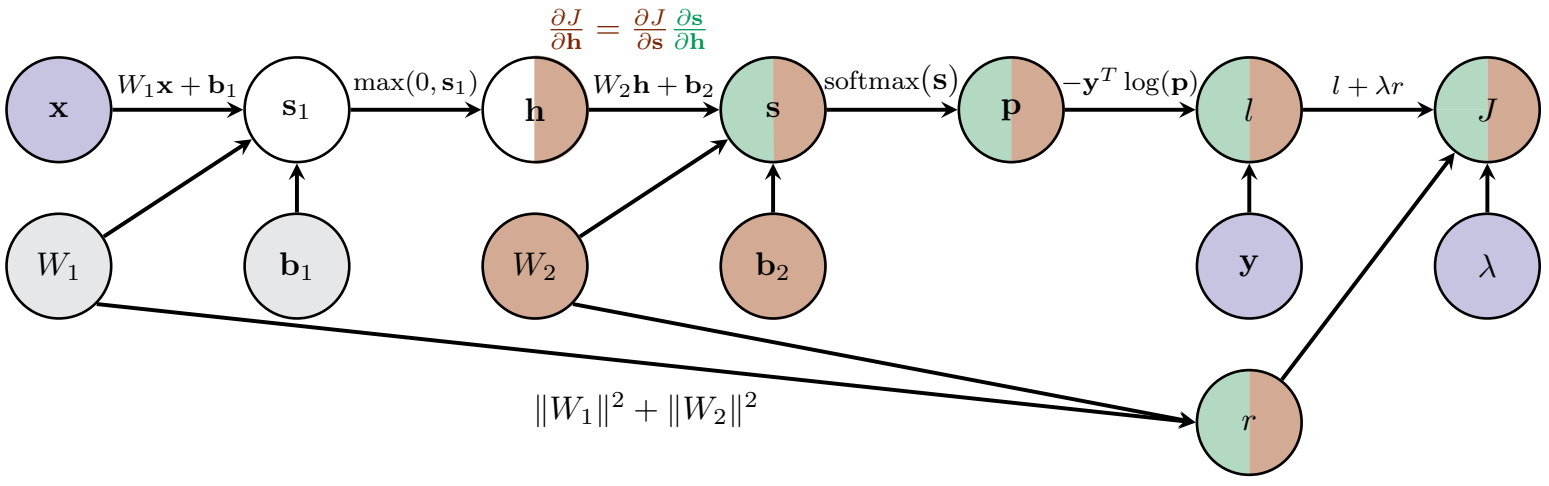
Compute local Jacobian of node $\mathbf{s}$ w.r.t. its parent $\mathbf{h}$



$$\boxed{\mathbf{s} = W_2\mathbf{h} + \mathbf{b}_2}$$

- The Jacobian we need to compute: $\frac{\partial \mathbf{s}}{\partial \mathbf{h}} = \begin{pmatrix} \frac{\partial s_1}{\partial h_1} & \cdots & \frac{\partial s_1}{\partial h_m} \\ \vdots & \vdots & \vdots \\ \frac{\partial s_c}{\partial h_1} & \cdots & \frac{\partial s_c}{\partial h_m} \end{pmatrix}$
- The individual derivatives: $\frac{\partial s_i}{\partial h_j} = W_{2,ij}$
- In vector notation: $\frac{\partial \mathbf{s}}{\partial \mathbf{h}} = W_2$

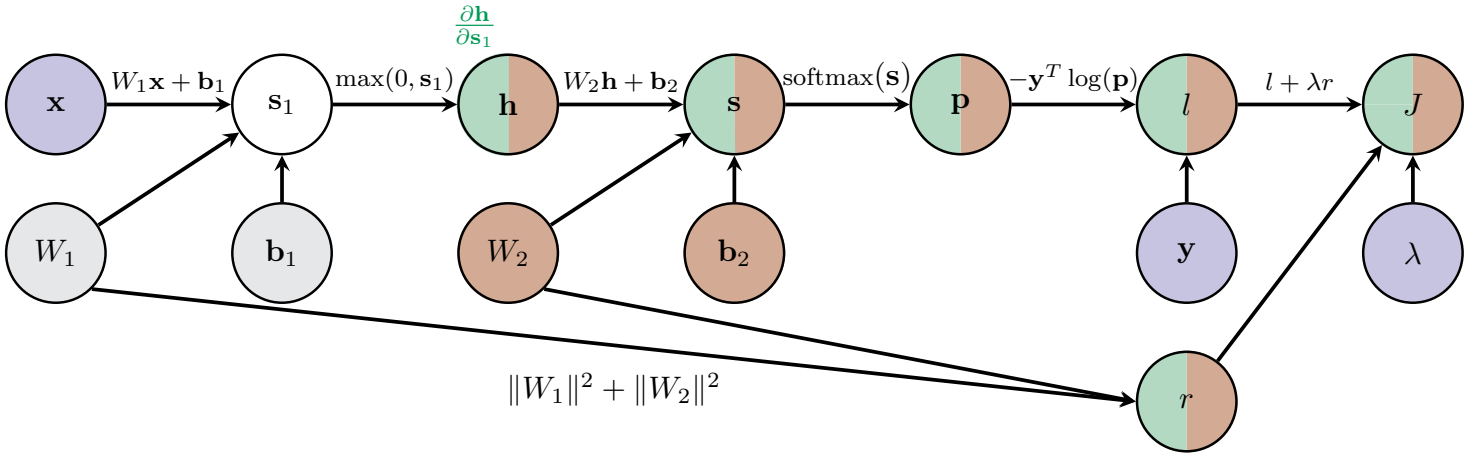
Compute gradient of J w.r.t. node $\mathbf{h}$



$$\boxed{\mathbf{s} = W_2\mathbf{h} + \mathbf{b}_2}$$

$$\frac{\partial J}{\partial \mathbf{h}} = \frac{\partial J}{\partial \mathbf{s}} \frac{\partial \mathbf{s}}{\partial \mathbf{h}}$$

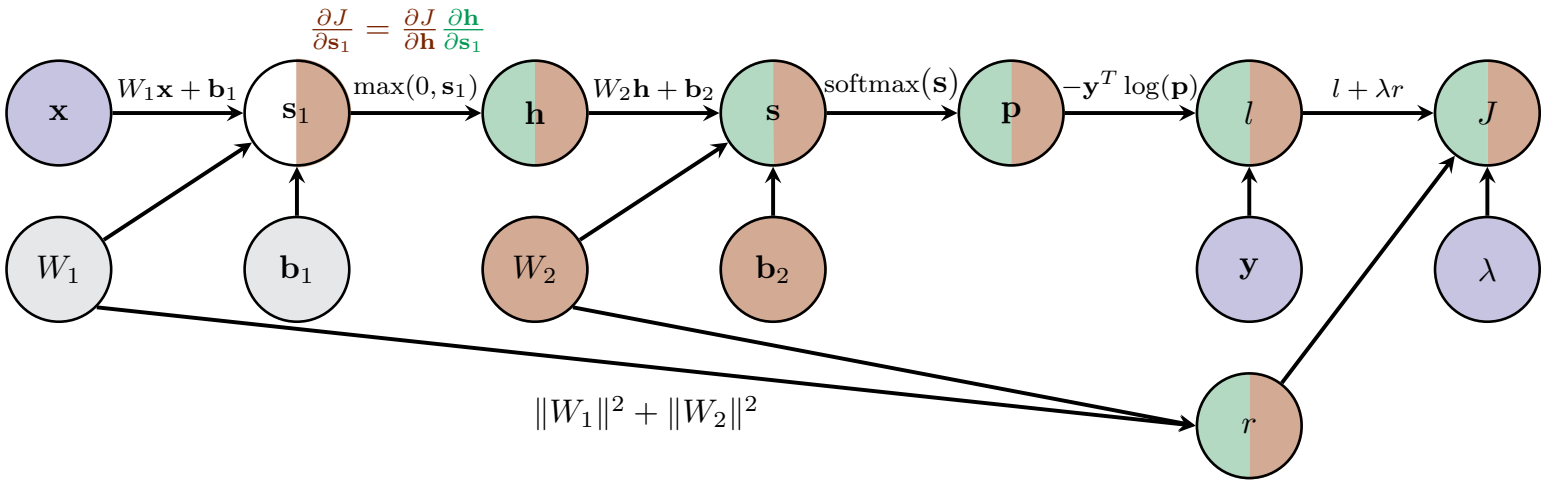
Compute local Jacobian of node $\mathbf{h}$ w.r.t. its parent $\mathbf{s}_1$



$$\boxed{\mathbf{h} = \max(0, \mathbf{s}_1)}$$

- The Jacobian we need to compute: $\frac{\partial \mathbf{h}}{\partial \mathbf{s}_1} = \begin{pmatrix} \frac{\partial h_1}{\partial s_{1,1}} & \cdots & \frac{\partial h_1}{\partial s_{1,m}} \\ \vdots & \vdots & \vdots \\ \frac{\partial h_m}{\partial s_{1,1}} & \cdots & \frac{\partial h_m}{\partial s_{1,m}} \end{pmatrix}$
- The individual derivatives: $\frac{\partial h_i}{\partial s_{1,j}} = \begin{cases} \mathrm{Ind}(s_{1,j} > 0) & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$
- In vector notation: $\frac{\partial \mathbf{h}}{\partial \mathbf{s}_1} = \mathrm{diag}(\mathrm{Ind}(\mathbf{s}_1 > 0))$

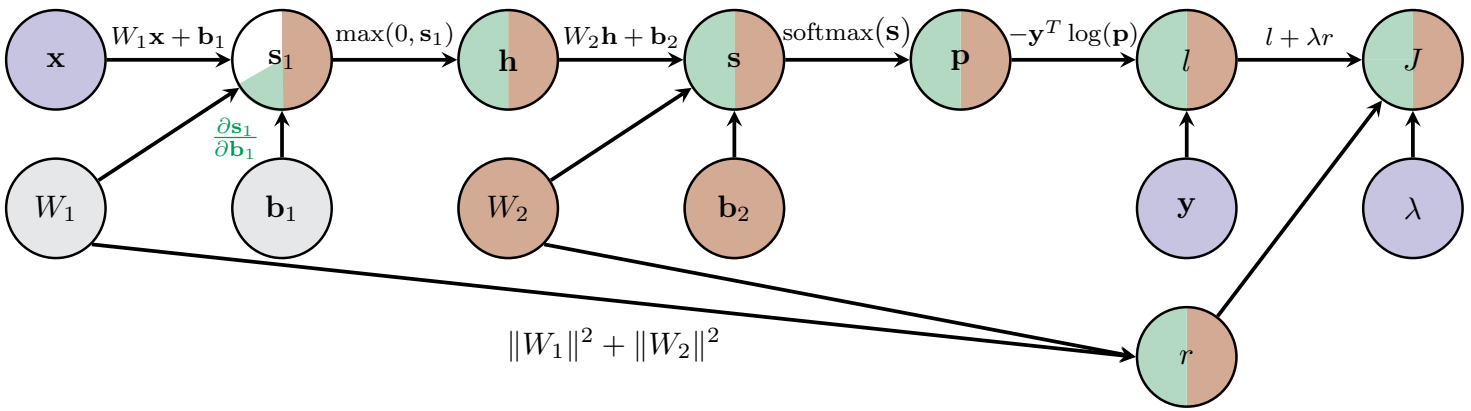
Compute gradient of J w.r.t. node $\mathbf{s}_1$



$$\mathbf{h} = \max(0, \mathbf{s}_1)$$

$$\frac{\partial J}{\partial \mathbf{s}_1} = \frac{\partial J}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{s}_1}$$

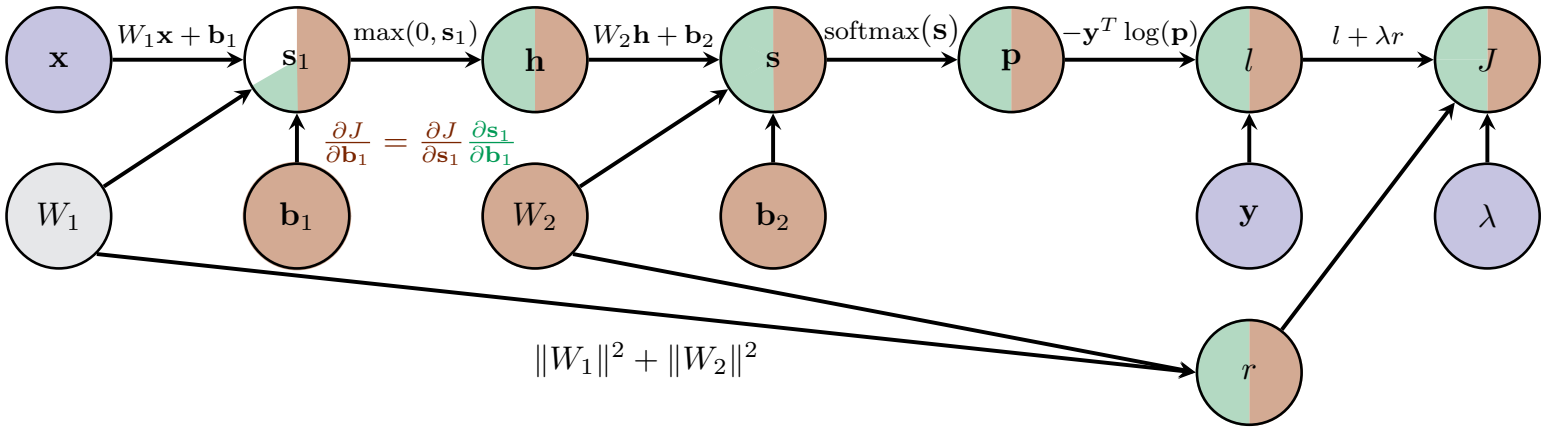
Compute local Jacobian of node $\mathbf{s}_1$ w.r.t. its parent $\mathbf{b}_1$



$$\boxed{\mathbf{s}_1 = W_1\mathbf{x} + \mathbf{b}_1}$$

- The Jacobian we need to compute: $\frac{\partial \mathbf{s}_1}{\partial \mathbf{b}_1} = \begin{pmatrix} \frac{\partial s_{1,1}}{\partial b_{1,1}} & \cdots & \frac{\partial s_{1,1}}{\partial b_{1,m}} \\ \vdots & \vdots & \vdots \\ \frac{\partial s_{1,m}}{\partial b_{1,1}} & \cdots & \frac{\partial s_{1,m}}{\partial b_{1,m}} \end{pmatrix}$
- The individual derivatives: $\frac{\partial s_{1,i}}{\partial b_{1,j}} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$
- In vector notation: $\frac{\partial \mathbf{s}_1}{\partial \mathbf{b}_1} = I_m$

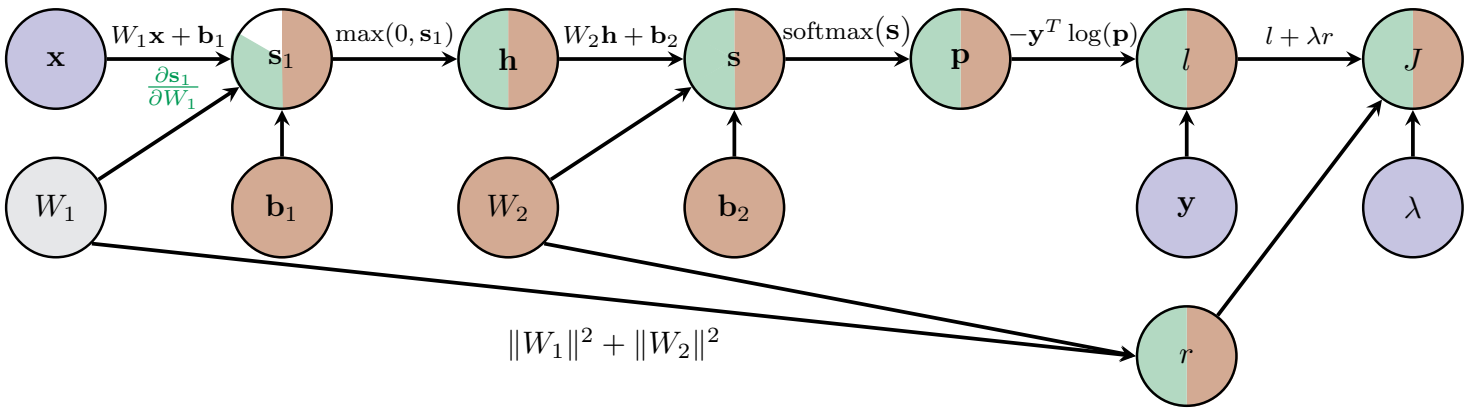
Compute gradient of J w.r.t. node $\mathbf{b}_1$



$$\boxed{\mathbf{s}_1 = W_1\mathbf{x} + \mathbf{b}_1}$$

$$\frac{\partial J}{\partial \mathbf{b}_1} = \frac{\partial J}{\partial \mathbf{s}_1} \frac{\partial \mathbf{s}_1}{\partial \mathbf{b}_1}$$

Compute local Jacobian of node $\mathbf{s}_1$ w.r.t. its parent W_1

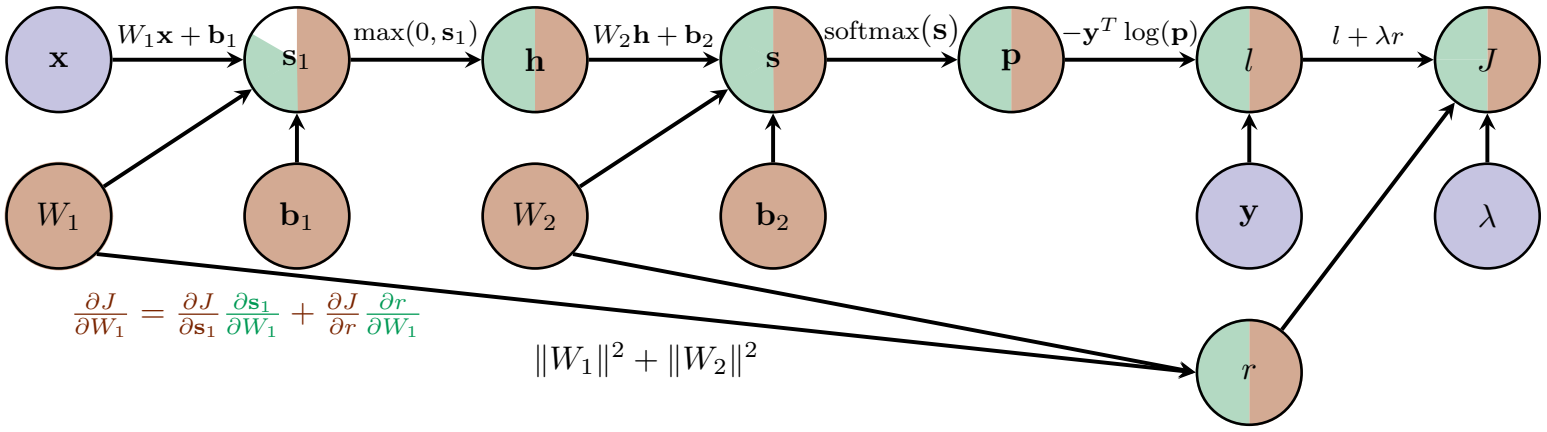


$$\mathbf{s}_1 = W_1\mathbf{x}+\mathbf{b}_1 = \left(I_m \otimes \mathbf{x}^T\right)\mathrm{vec}(W_1)$$

- Let $\mathbf{v} = \mathrm{vec}(W_1)$. Jacobian to compute: $\frac{\partial \mathbf{s}_1}{\partial \mathbf{v}} = \begin{pmatrix} \frac{\partial s_{1,1}}{\partial v_1} & \cdots & \frac{\partial s_{1,1}}{\partial v_{dm}} \\ \vdots & \vdots & \vdots \\ \frac{\partial s_{1,m}}{\partial v_1} & \cdots & \frac{\partial s_{1,m}}{\partial v_{dm}} \end{pmatrix}$
- The individual derivatives: $\frac{\partial s_{1,i}}{\partial v_j} = \begin{cases} x_{j-(i-1)d} & \text{if } (i-1)d+1 \le j \le id \\ 0 & \text{otherwise} \end{cases}$
- In vector notation: $\frac{\partial \mathbf{s}_1}{\partial \mathbf{v}} = I_m \otimes \mathbf{x}^T$

Backward Pass

Compute gradient of J w.r.t. node W_1



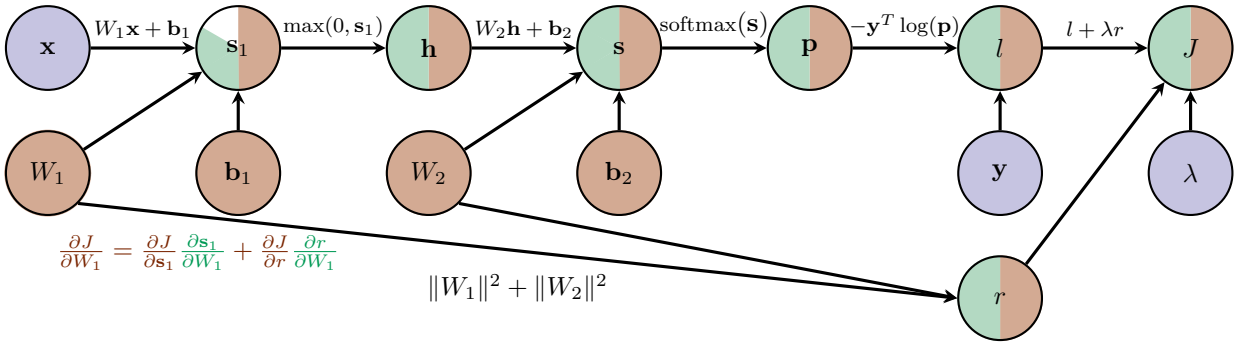
$$\mathbf{s}_1 = W_1\mathbf{x} + \mathbf{b}_1 = \left(I_m \otimes \mathbf{x}^T\right) \mathrm{vec}(W_1) + \mathbf{b}_1$$

$$\begin{aligned}
\frac{\partial J}{\partial \mathrm{vec}(W_1)} &= \frac{\partial J}{\partial \mathbf{s}_1}\frac{\partial \mathbf{s}_1}{\partial \mathrm{vec}(W_1)} + \frac{\partial J}{\partial r}\frac{\partial r}{\partial \mathrm{vec}(W_1)} \\
&= \begin{pmatrix} g_1\mathbf{x}^T & g_2\mathbf{x}^T & \cdots & g_m\mathbf{x}^T \end{pmatrix} + \lambda\, \mathrm{vec}(W_1)^T
\end{aligned}$$

← gradient needed for learning

if we set $\mathbf{g} = \frac{\partial J}{\partial \mathbf{s}_1}$.

Compute gradient of J w.r.t. node W_1



$$\mathbf{s}_1 = W_1\mathbf{x}+\mathbf{b}_1 = \left(I_m \otimes \mathbf{x}^T\right)\text{vec}(W_1)+\mathbf{b}_1$$

Can convert

$$\frac{\partial J}{\partial \text{vec}(W_1)} = \begin{pmatrix} g_1\mathbf{x}^T & g_2\mathbf{x}^T & \cdots & g_m\mathbf{x}^T \end{pmatrix} + 2\lambda\,\text{vec}(W_1)^T$$

(where $\mathbf{g} = \frac{\partial J}{\partial \mathbf{s}_1}$) from a vector $(1 \times md)$ back to a 2D matrix $(m \times d)$:

$$\frac{\partial J}{\partial W_1} = \begin{pmatrix} g_1\mathbf{x}^T \\ g_2\mathbf{x}^T \\ \vdots \\ g_C\mathbf{x}^T \end{pmatrix} + 2\lambda W_1 = \mathbf{g}^T\mathbf{x}^T + 2\lambda W_1$$

Aggregated backward pass for a 2-layer neural network

1. Let
$$\mathbf{g} = -(\mathbf{y} - \mathbf{p})^T$$
2. Gradient of J w.r.t. second bias vector is the $1 \times C$ vector
$$\frac{\partial J}{\partial \mathbf{b}_2} = \mathbf{g}$$
3. Gradient of J w.r.t. second weight matrix W_2 is the $C \times m$ matrix
$$\frac{\partial J}{\partial W_2} = \mathbf{g}^T \mathbf{h}^T + 2\lambda W_2$$
4. Back-propagate the gradient vector $\mathbf{g}$ to the first layers
$$\mathbf{g} = \mathbf{g} W_2$$
$$\mathbf{g} = \mathbf{g} \operatorname{diag}(\operatorname{Ind}(\mathbf{s}_1 > 0)) \leftarrow \text{assuming ReLu activation}$$
5. Gradient of J w.r.t. the first bias vector is the $1 \times d$ vector
$$\frac{\partial J}{\partial \mathbf{b}_1} = \mathbf{g}$$
6. Gradient of J w.r.t. the first weight matrix W_1 is the $m \times d$ matrix
$$\frac{\partial J}{\partial W_1} = \mathbf{g}^T \mathbf{x}^T + 2\lambda W_1$$

Gradient Computations for a mini-batch

2-layer scoring function + SoftMax + cross-entropy loss + Regularization

- Compute gradients of l w.r.t. $W_1, W_2, \mathbf{b}_1, \mathbf{b}_2$ for each $(\mathbf{x}, y) \in \mathcal{D}^{(t)}$:
  - Set all entries in $\frac{\partial L}{\partial \mathbf{b}_1}$, $\frac{\partial L}{\partial \mathbf{b}_2}$, $\frac{\partial L}{\partial W_1}$ and $\frac{\partial L}{\partial W_2}$ to zero.
  - for each $(\mathbf{x}, y) \in \mathcal{D}^{(t)}$
    1. Forward pass: Apply network, given current parameter values, to $\mathbf{x}$.
    2. Let $\mathbf{g} = -(\mathbf{y} - \mathbf{p})^T$
    3. Add gradient of l w.r.t. $\mathbf{b}_2$ & W_2 computed at $(\mathbf{x}, y)$
    $$\frac{\partial L}{\partial \mathbf{b}_2} \mathrel{+}= \mathbf{g}, \quad \frac{\partial L}{\partial W_2} \mathrel{+}= \mathbf{g}^T \mathbf{h}^T$$
    4. Back-propagate gradient through 2nd fully connected layer
    $$\mathbf{g} = \mathbf{g} W_2$$
    $$\mathbf{g} = \mathbf{g} \operatorname{diag}(\operatorname{Ind}(\mathbf{s}_1 > 0))$$
    5. Add gradient of l w.r.t. first layer parameters computed at $(\mathbf{x}, y)$
    $$\frac{\partial L}{\partial \mathbf{b}_1} \mathrel{+}= \mathbf{g}, \quad \frac{\partial L}{\partial W_1} \mathrel{+}= \mathbf{g}^T \mathbf{x}^T$$
  - Divide by the number of entries in $\mathcal{D}^{(t)}$:
  $$\frac{\partial L}{\partial W_i} \mathrel{/}= |\mathcal{D}^{(t)}|, \quad \frac{\partial L}{\partial \mathbf{b}_i} \mathrel{/}= |\mathcal{D}^{(t)}| \quad \text{for } i = 1, 2$$
- Add the gradient for the regularization term
$$\frac{\partial J}{\partial W_i} = \frac{\partial L}{\partial W_i} + 2\lambda W_i, \quad \frac{\partial J}{\partial \mathbf{b}_i} = \frac{\partial L}{\partial \mathbf{b}_i} \quad \text{for } i = 1, 2$$

Efficient computations for mini-batch gradient of loss

- Let $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_{n_b}, \mathbf{y}_{n_b})\}$ be the data in the mini-batch $\mathcal{D}^{(t)}$.

- **Organize the batch data**:

  Gather all $\mathbf{x}_i$'s from the batch into a matrix, similarly for $\mathbf{y}_i$'s

$$X_{\text{batch}} = \begin{pmatrix} \uparrow & & \uparrow \\ \mathbf{x}_1 & \cdots & \mathbf{x}_{n_b} \\ \downarrow & & \downarrow \end{pmatrix}, \quad Y_{\text{batch}} = \begin{pmatrix} \uparrow & & \uparrow \\ \mathbf{y}_1 & \cdots & \mathbf{y}_{n_b} \\ \downarrow & & \downarrow \end{pmatrix}$$

- **Forward Pass**:

$$H_{\text{batch}} = \max\left(W_1 X_{\text{batch}} + \mathbf{b}_1 \mathbf{1}_{n_b}^T, 0\right)$$

$$\mathbf{P}_{\text{batch}} = \text{SoftMax}\left(W_2 H_{\text{batch}} + \mathbf{b}_2 \mathbf{1}_{n_b}^T\right)$$

Efficient computations for mini-batch gradient of loss

Backward Pass

1. Set

$$G_{\text{batch}} = -\left(Y_{\text{batch}} - P_{\text{batch}}\right)$$

2. Then

$$\frac{\partial L}{\partial W_2} = \frac{1}{n_b} G_{\text{batch}} H_{\text{batch}}^T, \qquad \frac{\partial L}{\partial \mathbf{b}_2} = \frac{1}{n_b} G_{\text{batch}} \mathbf{1}_{n_b}$$

3. Propagate the gradient back through the second layer

$$G_{\text{batch}} = W_2^T G_{\text{batch}}$$

$$G_{\text{batch}} = G_{\text{batch}} \odot \text{Ind}\left(H_{\text{batch}} > 0\right)$$

4. Then

$$\frac{\partial L}{\partial W_1} = \frac{1}{n_b} G_{\text{batch}} X_{\text{batch}}^T, \qquad \frac{\partial L}{\partial \mathbf{b}_1} = \frac{1}{n_b} G_{\text{batch}} \mathbf{1}_{n_b}$$

Forward pass for a k-layer neural network

- Let $\mathbf{x}^{(0)} = \mathbf{x}$ represent the input.
- for $i = 1, \ldots, k-1$

$$\mathbf{s}^{(i)} = W_i \mathbf{x}^{(i-1)} + \mathbf{b}_i$$
$$\mathbf{x}^{(i)} = \max\left(0, \mathbf{s}^{(i)}\right)$$

- Apply the final linear transformation

$$\mathbf{s}^{(k)} = W_k \mathbf{x}^{(k-1)} + \mathbf{b}_k$$

- Apply SoftMax operation to turn final scores into probabilities

$$\mathbf{p} = \frac{\exp(\mathbf{s}^{(k)})}{\mathbf{1}^T \exp(\mathbf{s}^{(k)})}$$

- Apply cross-entropy loss and regularization to measure performance w.r.t. ground truth label $\mathbf{y}$

$$J = -\mathbf{y}^T \log(\mathbf{p}) + \lambda \sum_{i=1}^{k} \|W_i\|^2$$

Assumed ReLu is the activation function at each intermediary layer.

Aggregated Backward pass for a k-layer neural network

The gradient computation for one training example $(\mathbf{x}, y)$:

- Let

$$\mathbf{g} = -(\mathbf{y} - \mathbf{p})^T$$

- for $i = k, k-1, \ldots, 1$
  1. The gradient of J w.r.t. bias vector $\mathbf{b}_i$

$$\frac{\partial J}{\partial \mathbf{b}_i} = \mathbf{g}$$

  2. Gradient of J w.r.t. weight matrix W_i

$$\frac{\partial J}{\partial W_i} = \mathbf{g}^T \mathbf{x}^{(i-1)T} + 2\lambda W_i$$

  3. If $i > 1$ - Back-propagate the gradient vector $\mathbf{g}$ to the previous layer

$$\mathbf{g} = \mathbf{g} W_i$$
$$\mathbf{g} = \mathbf{g} \, \mathrm{diag}(\mathrm{Ind}(\mathbf{s}^{(i-1)} > 0))$$

Efficient computations for mini-batch gradient of loss

- Let $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_{n_b}, \mathbf{y}_{n_b})\}$ be the data in the mini-batch $\mathcal{D}^{(t)}$.

- **Organize the batch data**:
  Gather all $\mathbf{x}_i$'s from the batch into a matrix, similarly for $\mathbf{y}_i$'s

$$X_{\text{batch}} = \begin{pmatrix} \uparrow & & \uparrow \\ \mathbf{x}_1 & \cdots & \mathbf{x}_{n_b} \\ \downarrow & & \downarrow \end{pmatrix}, \quad Y_{\text{batch}} = \begin{pmatrix} \uparrow & & \uparrow \\ \mathbf{y}_1 & \cdots & \mathbf{y}_{n_b} \\ \downarrow & & \downarrow \end{pmatrix}$$

- **Forward Pass**: (let $X^{(0)}_{\text{batch}} = X_{\text{batch}}$)
  - for $l = 1, \ldots, k-1$

$$X^{(l)}_{\text{batch}} = \max\left(W_l X^{(l-1)}_{\text{batch}} + \mathbf{b}_l \mathbf{1}^T_{n_b}, 0\right)$$

  - Then

$$P_{\text{batch}} = \text{SoftMax}\left(W_k X^{(k-1)}_{\text{batch}} + \mathbf{b}_k \mathbf{1}^T_{n_b}\right)$$

Efficient computations for mini-batch gradient of loss

Backward Pass

1. Set

$$G_{\text{batch}} = -\left(Y_{\text{batch}} - P_{\text{batch}}\right)$$

2. Then

   for $l = k, k-1, \ldots, 2$

$$\frac{\partial L}{\partial W_l} = \frac{1}{n_b} G_{\text{batch}} {X_{\text{batch}}^{(l-1)}}^T, \quad \frac{\partial L}{\partial \mathbf{b}_l} = \frac{1}{n_b} G_{\text{batch}} \mathbf{1}_{n_b}$$

$$G_{\text{batch}} = W_l^T \, G_{\text{batch}}$$

$$G_{\text{batch}} = G_{\text{batch}} \odot \text{Ind}\left(X_{\text{batch}}^{(l-1)} > 0\right)$$

3. Finally

$$\frac{\partial L}{\partial W_1} = \frac{1}{n_b} G_{\text{batch}} X_{\text{batch}}^T, \quad \frac{\partial L}{\partial \mathbf{b}_1} = \frac{1}{n_b} G_{\text{batch}} \mathbf{1}_{n_b}$$

Training Neural Networks a little bit of history

A bit of history

- Perceptron algorithm invented by Frank Rosenblatt (1957).
- **Mark 1 Perceptron machine**

  First implementation of the perceptron algorithm.
- Machine was connected to camera producing 20×20 pixel image and recognized letters.
- Perceptron classification fn:

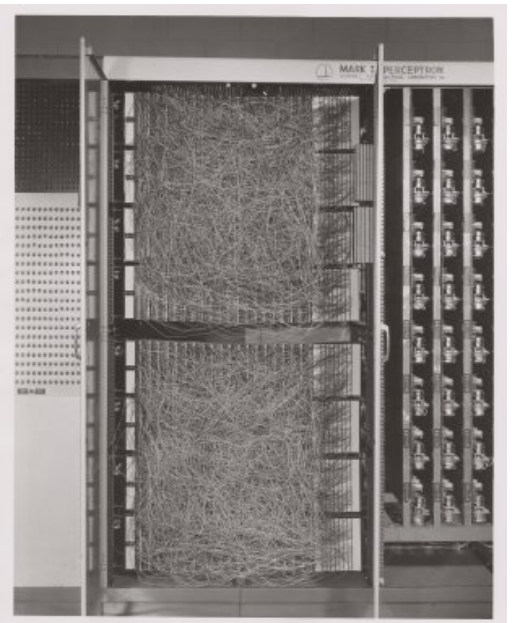
$$g(\mathbf{x}; \mathbf{w}, b) = \begin{cases} 1 & \text{if } \mathbf{w}^T\mathbf{x} + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

- For labelled training example $(\mathbf{x}, y)$ $(y \in \{-1, 1\})$ the Perceptron loss is

$$l_p(\mathbf{x}, y, \mathbf{w}, b) = \max(0, -y(\mathbf{w}^T\mathbf{x} + b))$$

- **Update rule**: Use SGD to learn $\mathbf{w}$. If training example $(\mathbf{x}_i, y_i)$ is incorrectly classified then

$$\mathbf{w} \leftarrow \mathbf{w} + y_i\mathbf{x}_i$$



Mark I

Perceptron machine

A bit of history

- ADALINE (Adaptive Linear Element) developed by **Widrow** and **Hoff** at Stanford in 1960.
- Adaline a single layer neural network with one output

$$\hat{y} = \mathbf{w}^T \mathbf{x} + b$$

- **Loss function**: for labelled training example $(\mathbf{x}, y)$

$$l(\mathbf{x}, y, \mathbf{w}, b) = \frac{1}{2}(y - (\mathbf{w}^T \mathbf{x} + b))^2 = \frac{1}{2}(y - \hat{y})^2$$

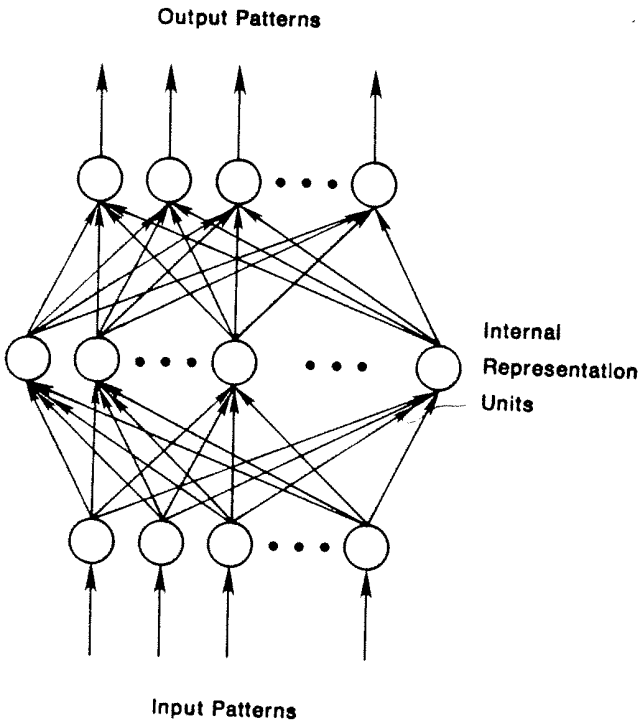
- **Update rule**: Use SGD with learning rate η to learn $\mathbf{w}$:

$$\mathbf{w} \leftarrow \mathbf{w} + \eta(y - \hat{y})\mathbf{x}$$

- Extension **Madaline**: a three-layer, fully connected, feed-forward artificial neural network architecture for classification.

A bit of history

Learning Internal Representations by Error Propagation, D. Rumelhart, G, Hinton and R. Williams, Parallel Distributed Processing: Explorations in the Microstructure of Cognition, 1986.



To be more specific, then, let

$$E_p = \frac{1}{2}\sum_j (t_{pj} - o_{pj})^2 \qquad (2)$$

be our measure of the error on input/output pattern p and let $E = \sum E_p$ *be our overall measure of the error. We wish to show that the delta rule implements a gradient descent in* E *when the units are linear. We will proceed by simply showing that*

$$-\frac{\partial E_p}{\partial w_{ji}} = \delta_{pj} i_{pi},$$

which is proportional to $\Delta_p w_{ji}$ *as prescribed by the delta rule. When there are no hidden units it is straightforward to compute the relevant derivative. For this purpose we use the chain rule to write the derivative as the product of two parts: the derivative of the error with respect to the output of the unit times the derivative of the output with respect to the weight.*

$$\frac{\partial E_p}{\partial w_{ji}} = \frac{\partial E_p}{\partial o_{pj}} \frac{\partial o_{pj}}{\partial w_{ji}}. \qquad (3)$$

The first part tells how the error changes with the output of the j*th unit and the second part tells how much changing* w_{ji} *changes that output. Now, the derivatives are easy to compute. First, from Equation 2*

$$\frac{\partial E_p}{\partial o_{pj}} = -(t_{pj} - o_{pj}) = -\delta_{pj}. \qquad (4)$$

Not surprisingly, the contribution of unit u_j *to the error is simply proportional to* δ_{pj}*. Moreover, since we have linear units,*

$$o_{pj} = \sum_i w_{ji} i_{pi}, \qquad (5)$$

from which we conclude that

$$\frac{\partial o_{pj}}{\partial w_{ji}} = i_{pi}.$$

Thus, substituting back into Equation 3, we see that

$$-\frac{\partial E_p}{\partial w_{ji}} = \delta_{pj} i_{pi} \qquad (6)$$

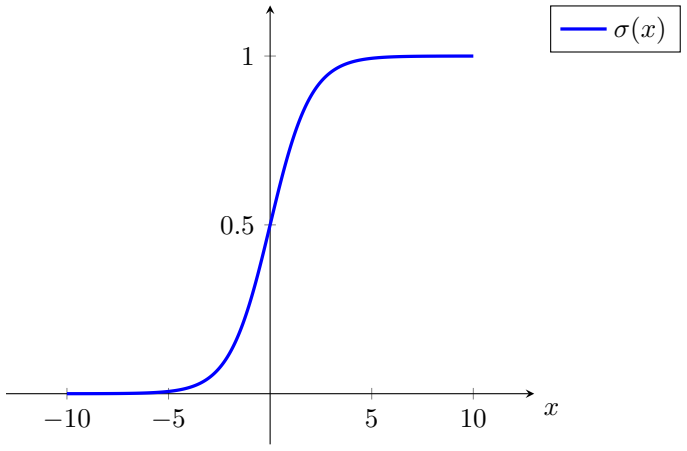
First time back-propagation became popular

Better understanding of gradient flow helped train deep networks with only BackProp

Understanding Effect of Activation Functions

Sigmoid

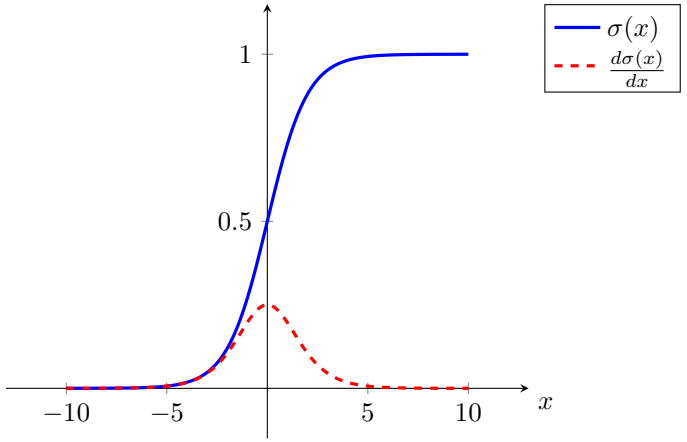
$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$



- Squashes numbers to range $[0, 1]$.
- Has nice interpretation as a saturating *firing rate* of a neuron.

Sigmoid

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$



Problems

1. Saturated activations **kill** the gradients.
   - Have a sigmoid activation

$$\mathbf{s} = W\mathbf{x} + \mathbf{b}$$
$$\mathbf{h} = \sigma(\mathbf{s})$$

   - Derivative of the sigmoid function is:

$$\frac{\partial h_i}{\partial s_i} = \frac{\exp(-s_i)}{(1 + \exp(-s_i))^2} \quad (= \sigma'(s_i) = \sigma(s_i)(1 - \sigma(s_i))$$

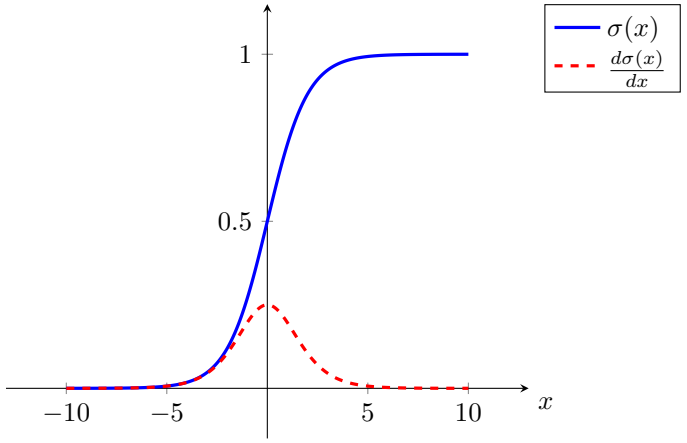
   - As

$$\frac{\partial J}{\partial s_i} = \frac{\partial J}{\partial h_i}\frac{\partial h_i}{\partial s_i} = \frac{\partial J}{\partial h_i}\,\sigma'(s_i)$$

What happens to $\partial J/\partial s_i$ when $|s_i| > 5$? Max value of $\sigma'(s_i)$?

Sigmoid

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$



Problems

1. Saturated activations **kill** the gradients.
2. Sigmoid outputs are not zero-centered.
   - Have a sigmoid activation

$$\mathbf{s} = W\mathbf{x} + \mathbf{b}$$
$$\mathbf{h} = \sigma(\mathbf{s})$$

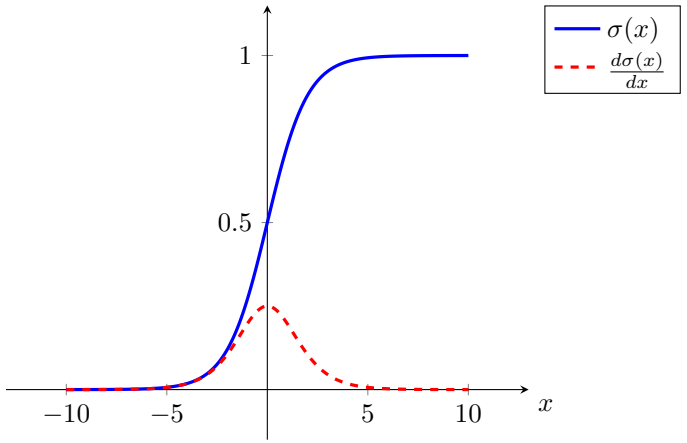
   - Then

$$\frac{\partial J}{\partial \mathbf{w}_i} = \frac{\partial J}{\partial h_i}\frac{\partial h_i}{\partial s_i}\frac{\partial s_i}{\partial \mathbf{w}_i} = \frac{\partial J}{\partial h_i}\,\sigma'(s_i)\,\mathbf{x}^T$$

What happens to $\frac{\partial J}{\partial \mathbf{w}_i}$ when all entries in $\mathbf{x}$ are positive?

Sigmoid

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$



Problems

1. Saturated activations **kill** the gradients.
2. Sigmoid outputs are not zero-centered.
   - Have a sigmoid activation

$$\mathbf{s} = W\mathbf{x} + \mathbf{b}, \quad \mathbf{h} = \sigma(\mathbf{s})$$

   - Then

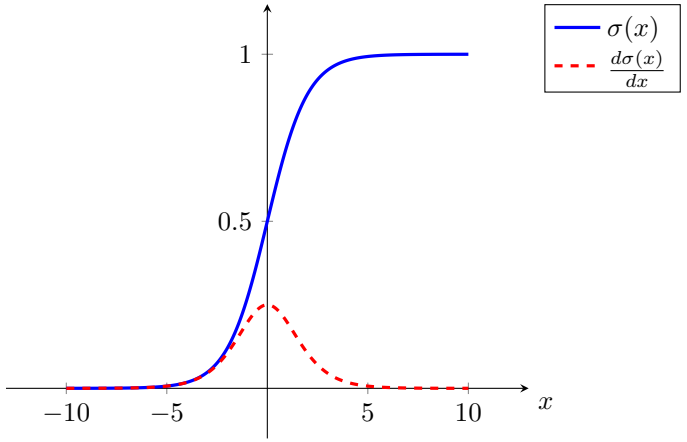
$$\frac{\partial J}{\partial \mathbf{w}_i} = \frac{\partial J}{\partial h_i}\frac{\partial h_i}{\partial s_i}\frac{\partial s_i}{\partial \mathbf{w}_i} = \underbrace{\frac{\partial J}{\partial h_i}}_{\text{positive or negative}} \underbrace{\sigma'(s_i)}_{\text{positive}} \underbrace{\mathbf{x}^T}_{\text{all positive}}$$

What happens to $\frac{\partial J}{\partial \mathbf{w}_i}$ when all entries in $\mathbf{x}$ are positive?

$\Longrightarrow$ entries of $\frac{\partial J}{\partial \mathbf{w}_i}$ are either all positive or all negative.

Sigmoid

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$



Problems

1. Saturated activations **kill** the gradients.
2. Sigmoid outputs are not zero-centered.
   - Have a sigmoid activation

     $$\mathbf{s} = W\mathbf{x} + \mathbf{b}, \quad \mathbf{h} = \sigma(\mathbf{s})$$

   - Then

     $$\frac{\partial J}{\partial \mathbf{w}_i} = \frac{\partial J}{\partial h_i}\frac{\partial h_i}{\partial s_i}\frac{\partial s_i}{\partial \mathbf{w}_i} = \frac{\partial J}{\partial \mathbf{w}_i} = \frac{\partial J}{\partial h_i}\frac{\partial h_i}{\partial s_i}\frac{\partial s_i}{\partial \mathbf{w}_i} = \frac{\partial J}{\partial h_i}\,\sigma'(s_i)\,\mathbf{x}^T$$

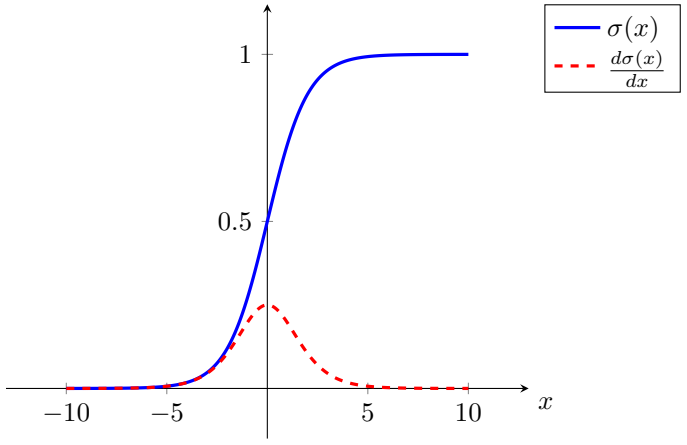
     What is $\frac{\partial J}{\partial \mathbf{w}_i}$ when all entries in $\mathbf{x}$ are $+$tive? (occurs after applying sigmoid)

     $\Longrightarrow$ entries of $\frac{\partial J}{\partial \mathbf{w}_i}$ are either all positive or all negative.

     $\Longrightarrow$ inefficient zig-zag update paths to find optimal $\mathbf{w}_i$

Sigmoid

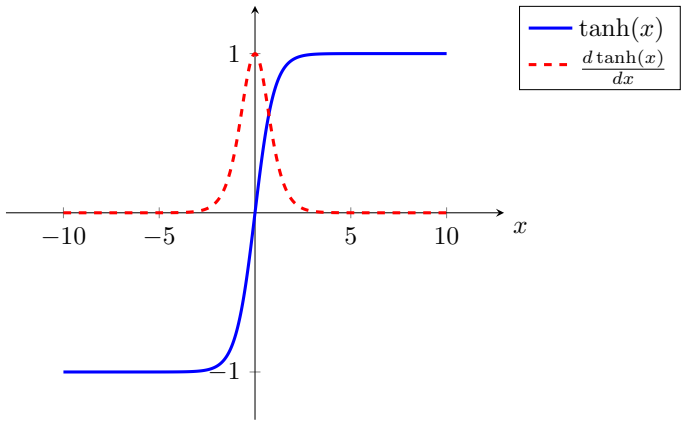
$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$



Problems

1. Saturated activations **kill** the gradients.
2. Sigmoid outputs are not zero-centered.
3. $\exp()$ is expensive to compute

$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

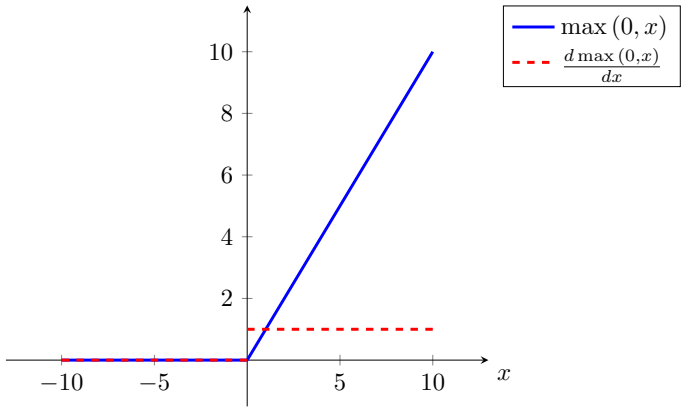


Properties

1. Squashes numbers to range $[-1, 1]$.
2. Tanh outputs are zero-centered.
3. Saturated activations kill the gradients.

Rectified Linear Unit (ReLu)

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

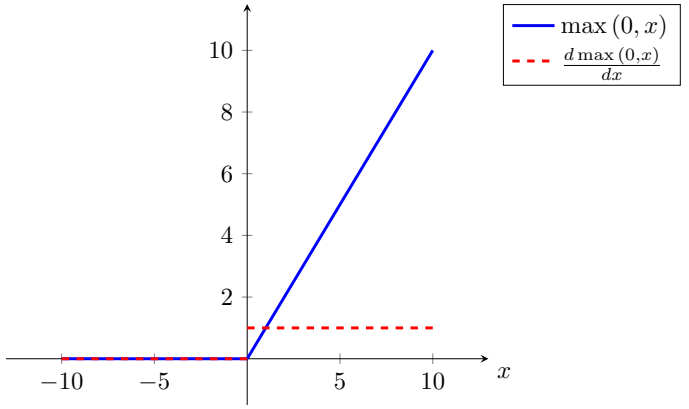


Pros

1. Does not saturate for large positive x.
2. Very computationally efficient.
3. In practice training of a ReLu network converges much faster than one with sigmoid/tanh activation functions.

Rectified Linear Unit (ReLu)

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



Problems

1. Output is not zero-centered
2. Negative inputs result in zero gradients.
   - Have a ReLu activation

$$\mathbf{s} = W\mathbf{x} + \mathbf{b}$$
$$\mathbf{h} = \max(0, \mathbf{s})$$

   - Derivative of the ReLu function is:

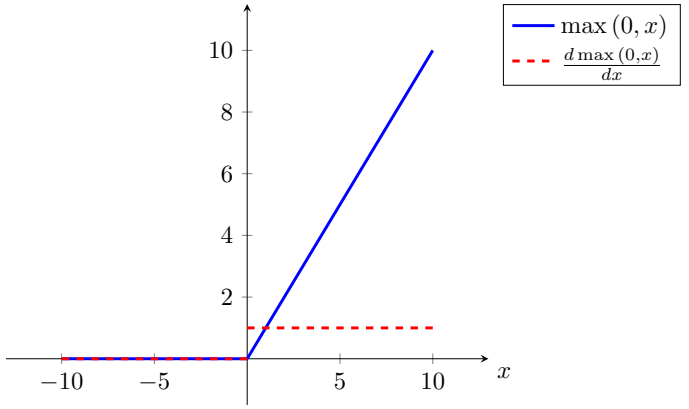
$$\frac{\partial h_i}{\partial s_j} = \begin{cases} 1 & \text{if } i = j \ \& \ s_j > 0 \\ 0 & \text{otherwise} \end{cases}$$

   - Then

$$\frac{\partial J}{\partial s_i} = \frac{\partial J}{\partial h_i}\frac{\partial h_i}{\partial s_i} = \begin{cases} \frac{\partial J}{\partial h_i} & \text{if } s_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

Rectified Linear Unit (ReLu)

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



Problems

1. Output is not zero-centered
2. Negative activations have zero gradients

   As

   $$\mathbf{s} = W\mathbf{x} + \mathbf{b}, \quad \mathbf{h} = \max(0, \mathbf{s})$$

   then

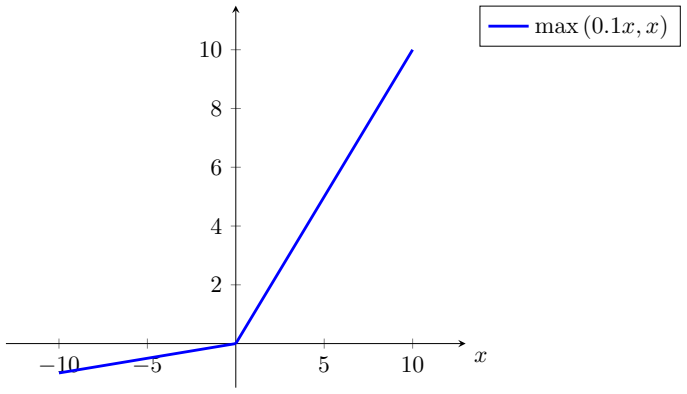
   $$\frac{\partial J}{\partial \mathbf{w}_i} = \frac{\partial J}{\partial h_i}\frac{\partial h_i}{\partial s_i}\frac{\partial s_i}{\partial \mathbf{w}_i} = \begin{cases} \frac{\partial J}{\partial h_i}\,\mathbf{x}^T & \text{if } s_i > 0 \\ \mathbf{0} & \text{otherwise} \end{cases}$$

   $\implies$ if $s_i < 0$ then $\mathbf{x}$ does not contribute to update of $\mathbf{w}_i$.
3. If $\forall \mathbf{x}$ have $s_i < 0$ then have a "dead neuron" and $\mathbf{w}_i$ is not up-dated.

   typically happens when b_i has large negative value

Leaky ReLu

Leaky ReLu$(x) = \max(.01x, x)$



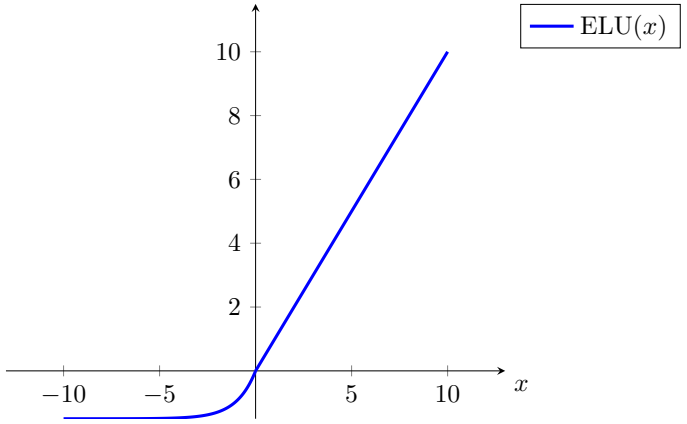
Pros

1. Does not saturate.
2. Computationally efficient.
3. In practice training of a Leaky ReLu network converges much faster than one with sigmoid/tanh activation functions.
4. Activations do not die.

[Mass et al., 2013] [He et al., 2015]

Exponential Linear Units (ELU)

$$\text{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(\exp(x) - 1) & \text{otherwise} \end{cases}$$



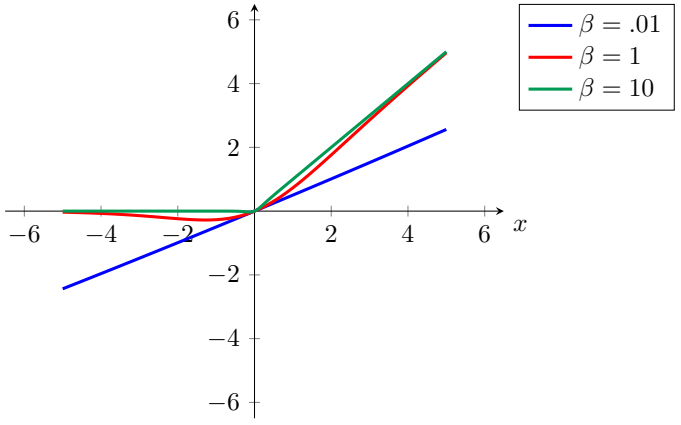
Pros & Cons

1. All the benefits of ReLu.
2. Activations do not die.
3. Closer to zero mean outputs.
4. Computation requires $\exp()$

[Clevert et al., 2015]

Swish

$$\text{Swish}_\beta(x) = x\,\sigma(\beta x)$$



Pros & Cons

1. Automatic search of possible activation functions by combining multiple *core functions* using reinforcement and local exhaustive search.

2. Swish outperformed all other options for Cifar-10 accuracy

3. Can train β, but as a default $\beta = 1$ works well

[Ramachandran et al., 2018]

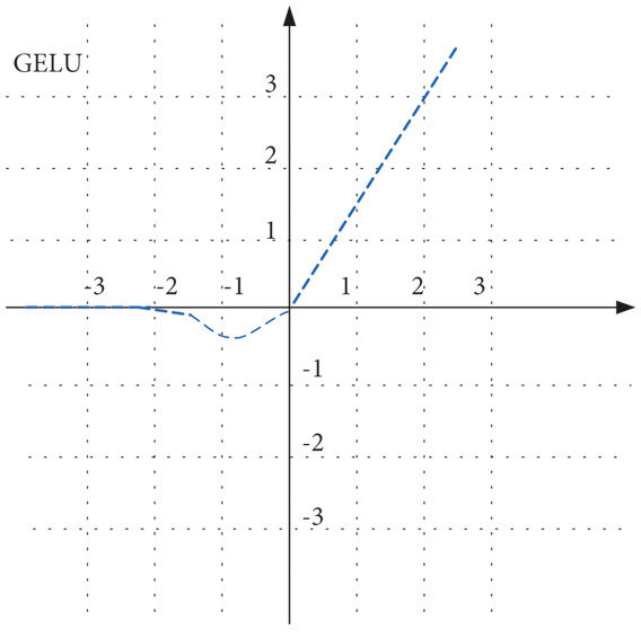
Which Activation Function?

In practice

- Use ReLU.
  - Be careful with your learning rates.
  - Initialize bias vectors to be slightly positive.
- Try out Leaky ReLU / ELU / Swish.
  - To squeeze out some marginal gains
- Try out tanh but don't expect much.
- Don't use sigmoid.

For Transformer network there are other common activation functions

Popular in Transformers



Gaussian Error Linear Unit:

$$\mathsf{GELU}(x) = xP(X \leq x) = x\,\Phi(x) = .5x\left(1 + \mathsf{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$$

Gated linear Units & Variants

- Input is a vector $\mathbf{x}$.
- Functions are applied independently per dimension.
- Applied at the fully-connected connections within the Transformer

$$\mathsf{GLU}(\mathbf{x}, W, V, b, c) = \sigma(\mathbf{x}W + b) \odot (\mathbf{x}V + c)$$

$$\mathsf{ReGLU}(\mathbf{x}, W, V, b, c) = \max(0, \mathbf{x}W + b) \odot (\mathbf{x}V + c)$$

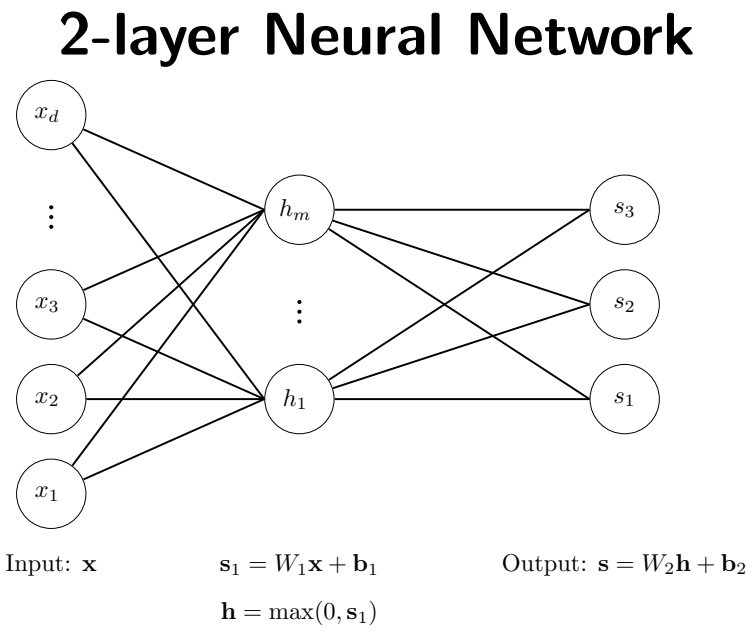
$$\mathsf{GEGLU}(\mathbf{x}, W, V, b, c) = \mathsf{GELU}(\mathbf{x}W + b) \odot (\mathbf{x}V + c)$$

$$\mathsf{SwiGLU}_\beta(\mathbf{x}, W, V, b, c) = \mathsf{Swish}_\beta(\mathbf{x}W + b) \odot (\mathbf{x}V + c)$$

W, V, b, c are parameters learned during training.

Effect of weight initialization & activation function on gradient flow

Pathological weight initialization



What happens when you initialize each weight matrix entry to the same constant? (each $W_{i,lm} = c$)

Pathological weight initialization example

Have

- L layers,
- C outputs,
- d dimensional inputs,
- d hidden nodes at each layer,
- no bias vectors and
- initialize weights with the same constant value

After mini-batch gd update step at time t the weight matrices have the form:

- W_L: each row is a different constant vector
- W_1: each column is a different constant vector
- For $2 \leq l \leq L-1$, W_l: has the form $a_t \mathbf{1}_d \mathbf{1}_d^T$.

Initialize with small random numbers

$$W_{i,lm} \sim N(w; 0, .01^2)$$

What happens in this case?

Initialize with small random numbers

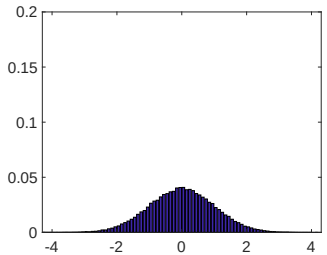
$$W_{i,lm} \sim N(w; 0, .01^2)$$

What happens in this case?

Works okay for small networks, but can lead to non-homogeneous distributions of activations across the layers of a deep network.

Consider this simple example

- **Input data**
  - Generate random input data, $N(0, 1^2)$, with $d = 500$.



Histogram of values in the input vectors.

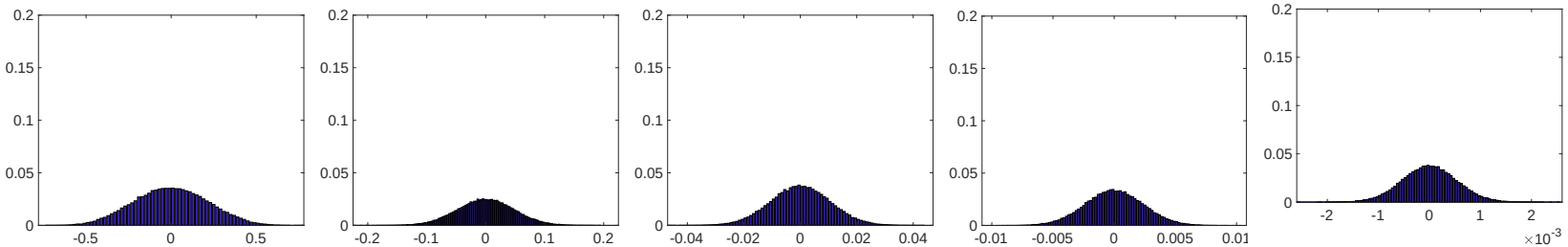
- **Network architecture**
  - Initialize a 10-layer network with 500 nodes at each layer.
  - Use a $\tanh$ activation function at each layer.

- **Weight initialization with small values**
  - Initialize weights according to:

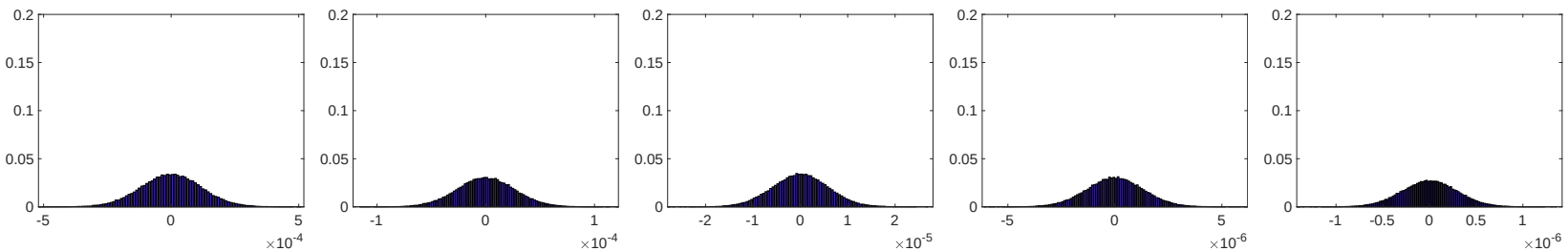
$$W_{i,lm} \sim N(w; 0, \sigma^2) \quad \text{where } \sigma = .01$$

Effect of initialization with small values on activations

- Initialize a 10-layer network with 500 nodes at each layer.
- Use $\tanh$ activation function at each layer.
- Initialize weight parameters with **small** values: $W_{i,lm} \sim N(w; 0, .01^2)$.



Layer 1 **Layer 2** **Layer 3** **Layer 4** **Layer 5**



Layer 6 **Layer 7** **Layer 8** **Layer 9** **Layer 10**

Histograms of activations at each layer.

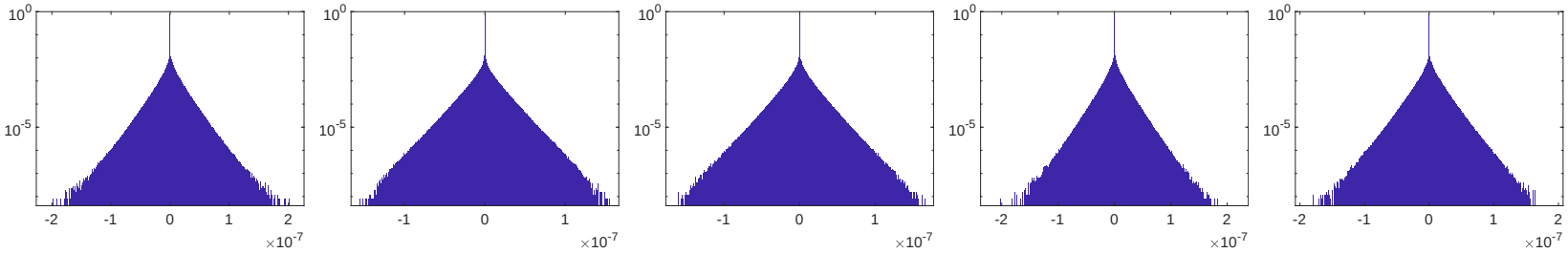
Note: values of x-axis rapidly decrease with each layer.

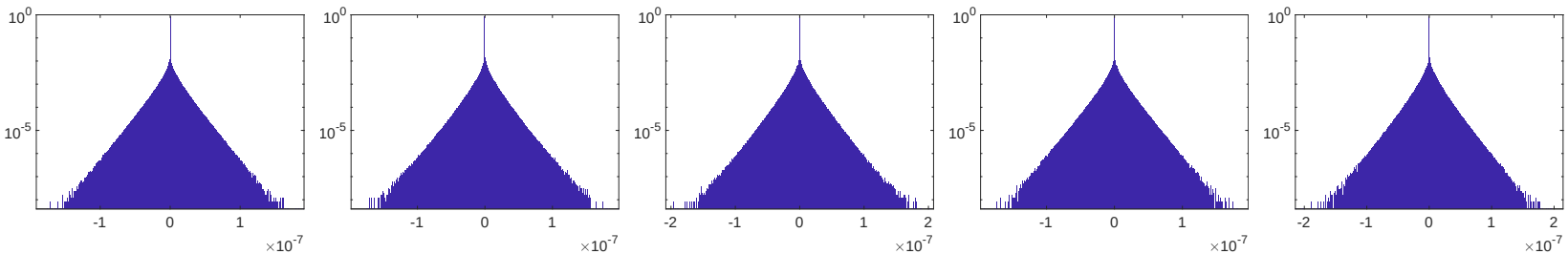
Effect of initialization with small values on gradients

- **Set-up**: $W_{i,lm} \sim N(w; 0, .01^2)$, $\tanh$ activation.
- Activations at each layer are increasingly small.
- What are the gradient values computed by back-prop algorithm?

Effect of initialization with small values on gradients

- **Set-up**: $W_{i,lm} \sim N(w; 0, .01^2)$, $\tanh$ activation.
- Activations at each layer are increasingly small.
- What are the gradient values computed by back-prop algorithm?

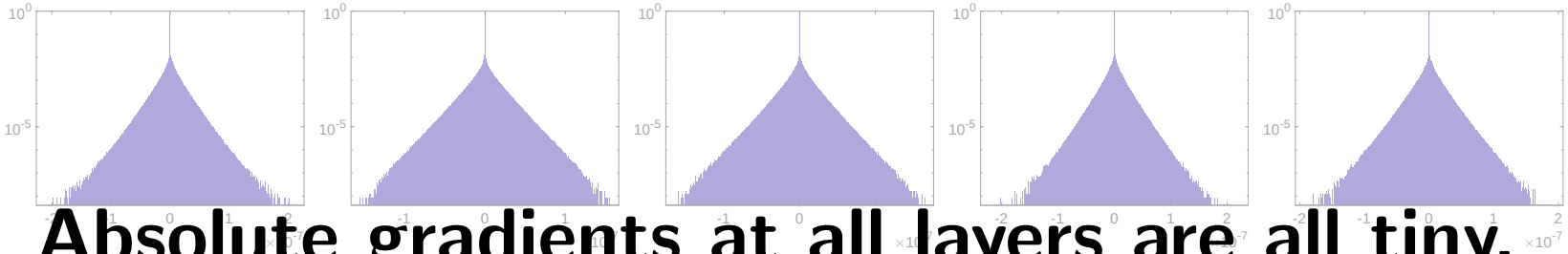




Histograms of gradients at each layer. (y-axis shown on logarithmic scale)

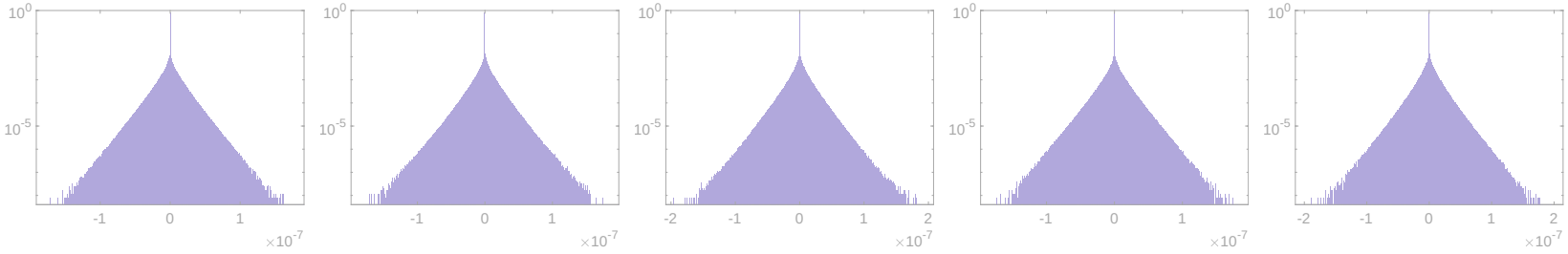
Effect of initialization with small values on gradients

- **Set-up**: $W_{i,lm} \sim N(w; 0, .01^2)$, $\tanh$ activation.
- Activations at each layer are increasingly small.
- What are the gradient values computed by back-prop algorithm?



Absolute gradients at all layers are all tiny.

Layer 1 Layer 2 Layer 3 Layer 4 Layer 5



Layer 6 Layer 7 Layer 8 Layer 9 Layer 10

Histograms of gradients at each layer. (y-axis shown on logarithmic scale)

Aggregated Backward pass for a k-layer neural network

The gradient computation for one training example $(\mathbf{x}, y)$:

- Let

$$\mathbf{g} = -(\mathbf{y} - \mathbf{p})^T$$

- for $i = k, k-1, \ldots, 1$
  1. The gradient of l w.r.t. bias vector $\mathbf{b}_i$

$$\frac{\partial l}{\partial \mathbf{b}_i} = \mathbf{g}$$

  2. Gradient of l w.r.t. weight matrix W_i

$$\frac{\partial l}{\partial W_i} = \mathbf{g}^T \mathbf{x}^{(i-1)T}$$

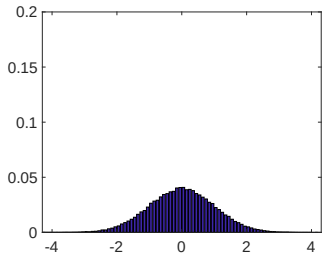
  3. Propagate the gradient vector $\mathbf{g}$ to the previous layer (if $i > 1$)

$$\mathbf{g} = \mathbf{g} W_i$$

$$\mathbf{g} = \mathbf{g} \operatorname{diag}(\tanh'(\mathbf{s}^{(i)}))$$

Change the initialization to bigger random numbers

- **Input data**
  - Generate random input data, $N(0, 1^2)$, with $d = 500$.



Histogram of values in the input vectors.

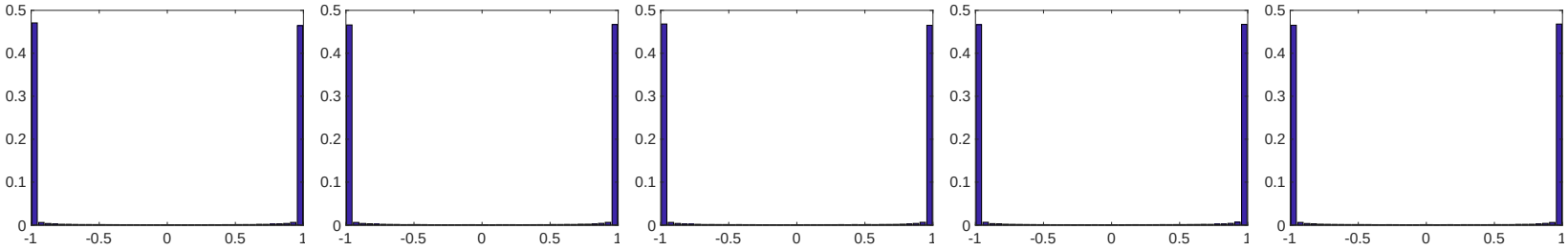
- **Network architecture**
  - Initialize a 10-layer network with 500 nodes at each layer.
  - Use a $\tanh$ activation function at each layer.

- **Weight initialization with large values**
  - Initialize weights according to:

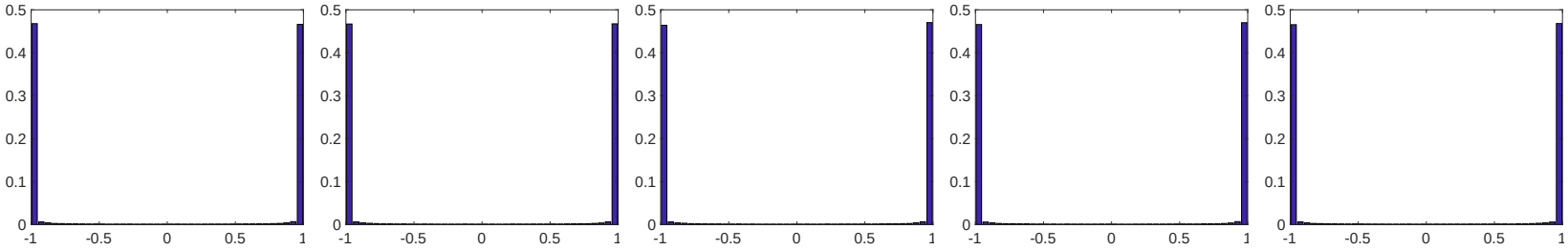
$$W_{i,lm} \sim N(w; 0, \sigma^2) \quad \text{where } \sigma = 1$$

Effect of initialization with large values on activations

- Initialize a 10-layer network with 500 nodes at each layer.
- Use $\tanh$ activation function at each layer.
- Initialize weight parameters with **large** values: $W_{i,lm} \sim N(w; 0, 1^2)$.



Layer 1 **Layer 2** **Layer 3** **Layer 4** **Layer 5**



Layer 6 **Layer 7** **Layer 8** **Layer 9** **Layer 10**

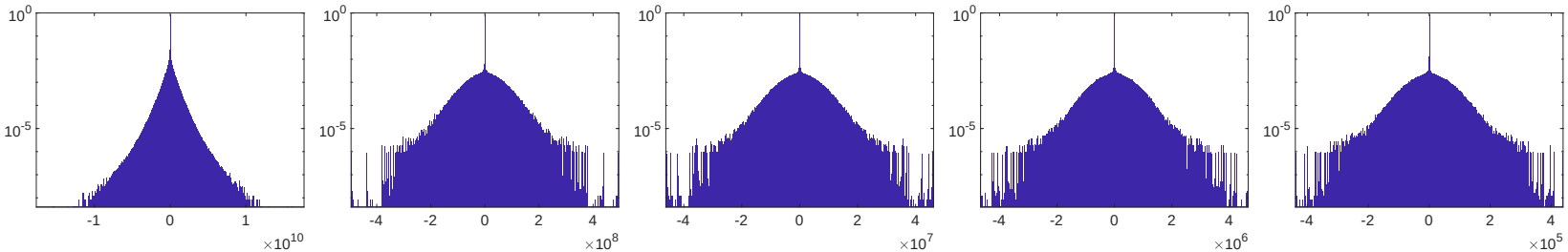
Histograms of activations at each layer.

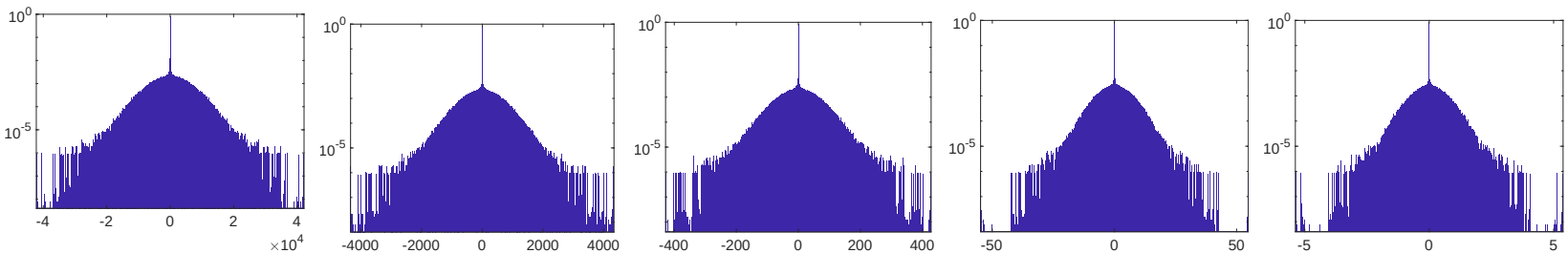
Effect of initialization with large values on gradients

- **Set-up**: $W_{i,lm} \sim N(w; 0, 1^2)$, $\tanh$ activation.
- Almost all neurons completely saturated, either -1 or $+1$.
- What are the gradient values computed by back-prop algorithm?

Effect of initialization with large values on gradients

- **Set-up**: $W_{i,lm} \sim N(w; 0, 1^2)$, $\tanh$ activation.
- Almost all neurons completely saturated, either -1 or $+1$.
- What are the gradient values computed by back-prop algorithm?



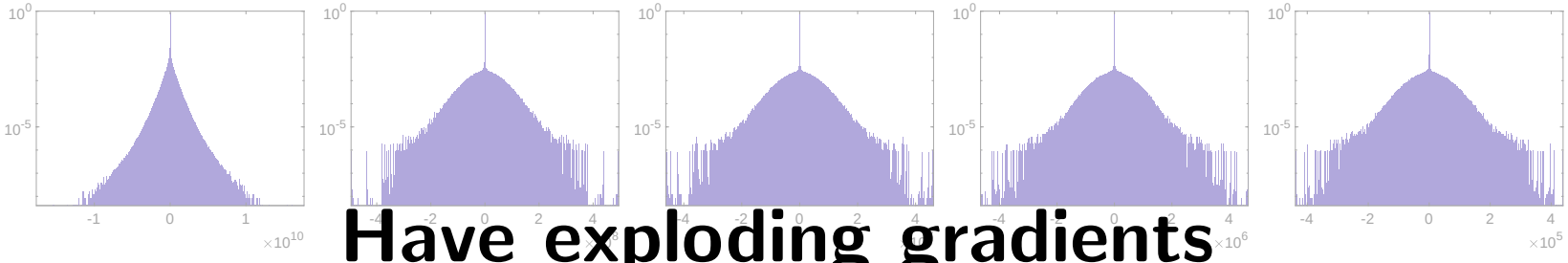


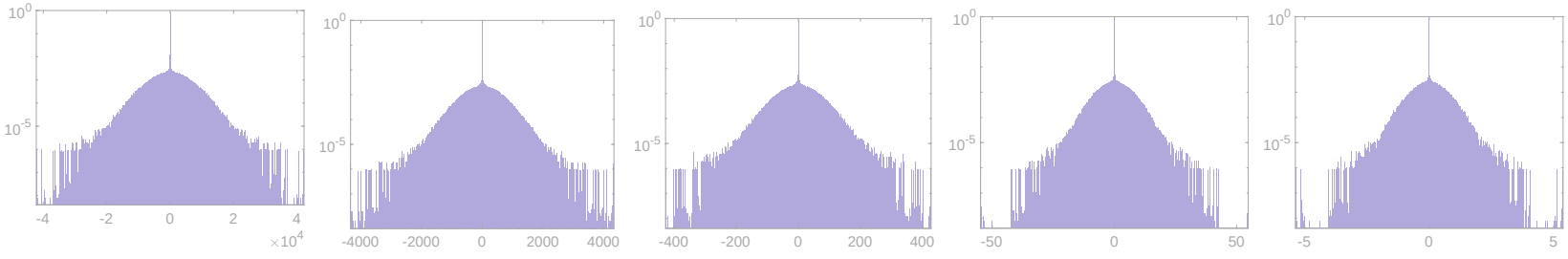
Histograms of gradients at each layer. (y-axis shown on logarithmic scale)

Note: the increase in the magnitude of the x-values as the layers decrease

Effect of initialization with large values on gradients

- **Set-up**: $W_{i,lm} \sim N(w; 0, 1^2)$, $\tanh$ activation.
- Almost all neurons completely saturated, either -1 or $+1$.
- What are the gradient values computed by back-prop algorithm?





Histograms of gradients at each layer. (y-axis shown on logarithmic scale)

Note: the increase in the magnitude of the x-values as the layers decrease

Aggregated Backward pass for a k-layer neural network

The gradient computation for one training example $(\mathbf{x}, y)$:

- Let

$$\mathbf{g} = -(\mathbf{y} - \mathbf{p})^T$$

- for $i = k, k-1, \ldots, 1$
  1. The gradient of l w.r.t. bias vector $\mathbf{b}_i$

$$\frac{\partial l}{\partial \mathbf{b}_i} = \mathbf{g}$$

  2. Gradient of l w.r.t. weight matrix W_i

$$\frac{\partial l}{\partial W_i} = \mathbf{g}^T \mathbf{x}^{(i-1)T}$$

  3. Propagate the gradient vector $\mathbf{g}$ to the previous layer (if $i > 1$)

$\mathbf{g} = \mathbf{g} W_i$ ← multiply by W_i which has many values > 1

$\mathbf{g} = \mathbf{g}\ \mathrm{diag}(\tanh'(\mathbf{s}^{(i)}))$ ← multiply each element of $\mathbf{g}$ by $0 < \tanh'(\mathbf{s}^{(i)}) \leq 1$

Increase by multiplication by W_i dominates dampening by $\mathrm{diag}(\tanh'(\mathbf{s}^{(i)}))$

Do I get similar problems, in this scenario, if I use ReLu activation functions instead of $\tanh$?

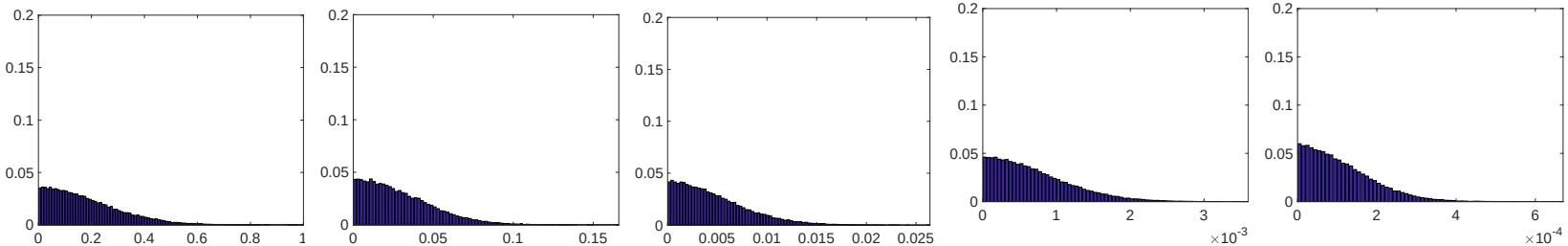
Yes. But some qualitative differences.

Effect of parameter init. on network with ReLu activations

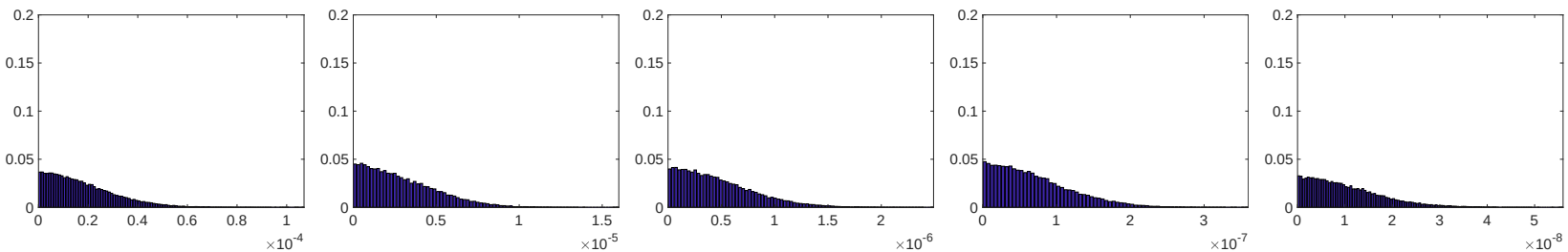
- **Effect on activations at each layer**
  - Regardless of value of σ at each layer many activations are exactly zero (especially when have zero bias vector).
  - Too small $\sigma \implies$ non-zero activations become increasingly close to zero as layer number increases.
  - Too large $\sigma \implies$ non-zero activations explode as layer number increases.
- **Effect on gradients at each layer**
  - Regardless of value of σ at each layer many gradients are exactly zero (especially when have zero bias vector).
  - Too small $\sigma \implies$ non-zero gradients are close to zero if network has many layers.
  - Too large $\sigma \implies$ non-zero gradients are very large if network has many layers.
  - Range of non-zero gradient values remains $\approx$ constant across layers.

Effect of initialization with small values on activations

- Initialize a 10-layer network with 500 nodes at each layer.
- Use a **ReLu activation function** at each layer.
- Initialize weight parameters with **small** values: $W_{i,lm} \sim N(w; 0, .01^2)$.



Layer 1 **Layer 2** **Layer 3** **Layer 4** **Layer 5**



Layer 6 **Layer 7** **Layer 8** **Layer 9** **Layer 10**

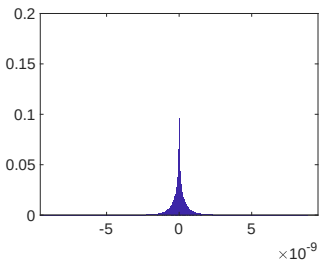
Histograms of **non-zero** activations at each layer.

(∼50% of activations are zero at each layer)

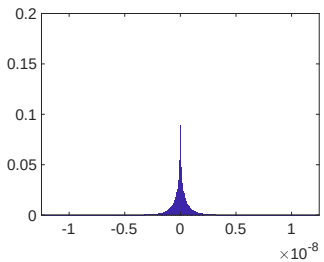
Note: values of x-axis rapidly decrease with each layer.

Effect of initialization with small values on gradients

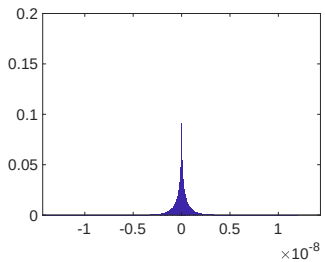
- **Set-up**: $W_{i,lm} \sim N(w; 0, .01^2)$, ReLu activation.
- Activations at each layer are increasingly small.
- What are the gradient values computed by back-prop algorithm?



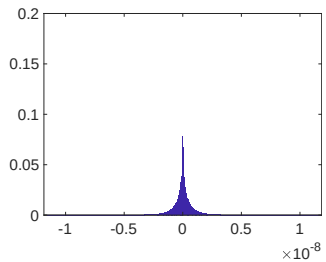
Layer 1



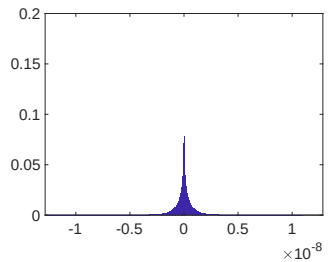
Layer 2



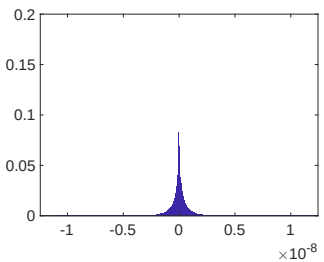
Layer 3



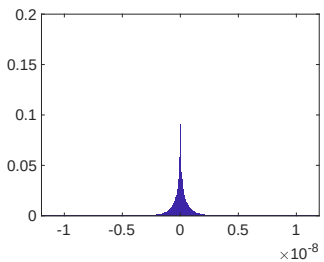
Layer 4



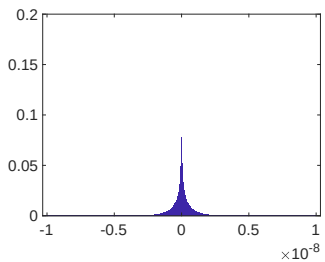
Layer 5



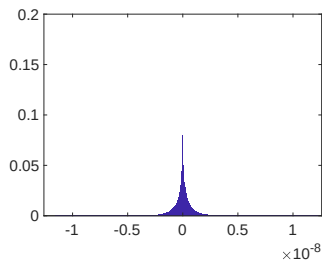
Layer 6



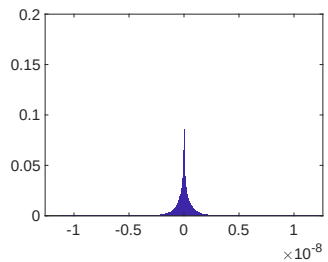
Layer 7



Layer 8



Layer 9



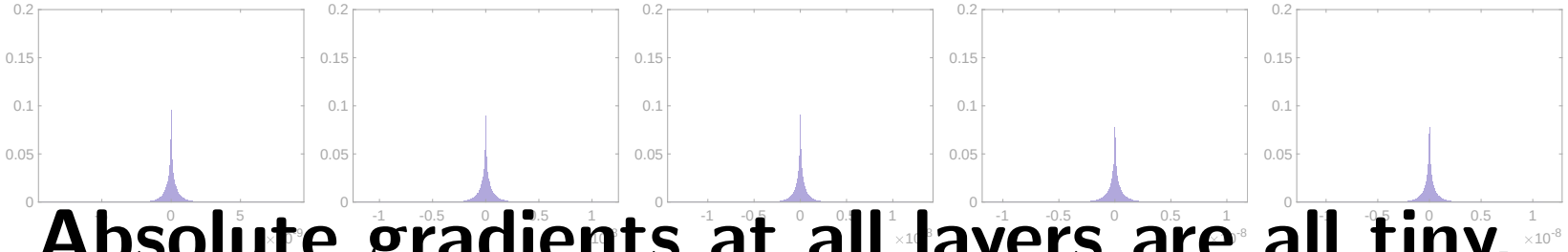
Layer 10

Histograms of **non-zero** gradients at each layer.

(∼90-95% of activations are zero at each layer, slight artifact of the toy input)

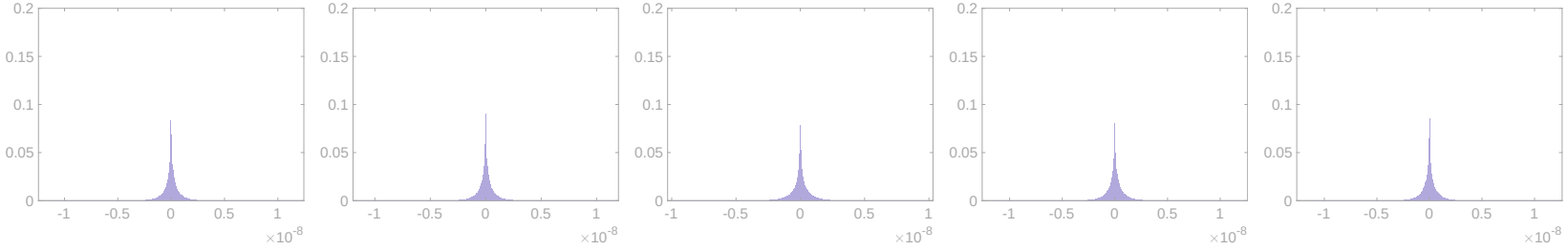
Effect of initialization with small values on gradients

- **Set-up**: $W_{i,lm} \sim N(w; 0, .01^2)$, ReLu activation.
- Activations at each layer are increasingly small.
- What are the gradient values computed by back-prop algorithm?



Absolute gradients at all layers are all tiny.

Layer 1 Layer 2 Layer 3 Layer 4 Layer 5



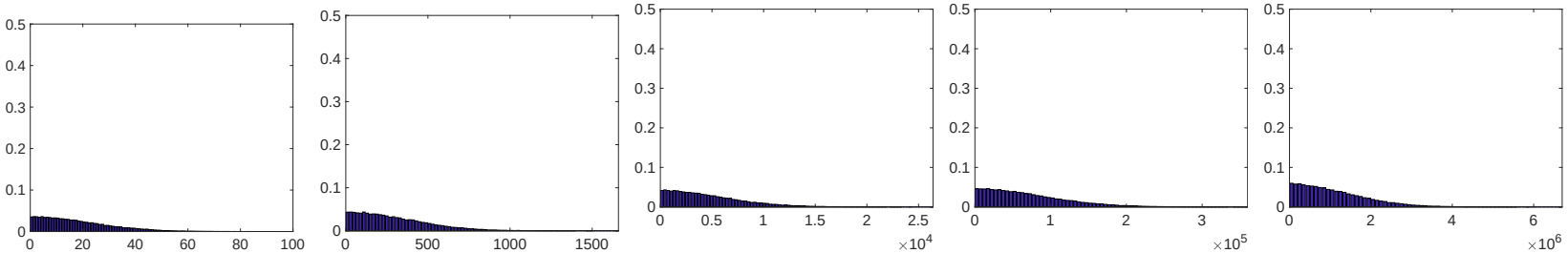
Layer 6 Layer 7 Layer 8 Layer 9 Layer 10

Histograms of **non-zero** gradients at each layer.

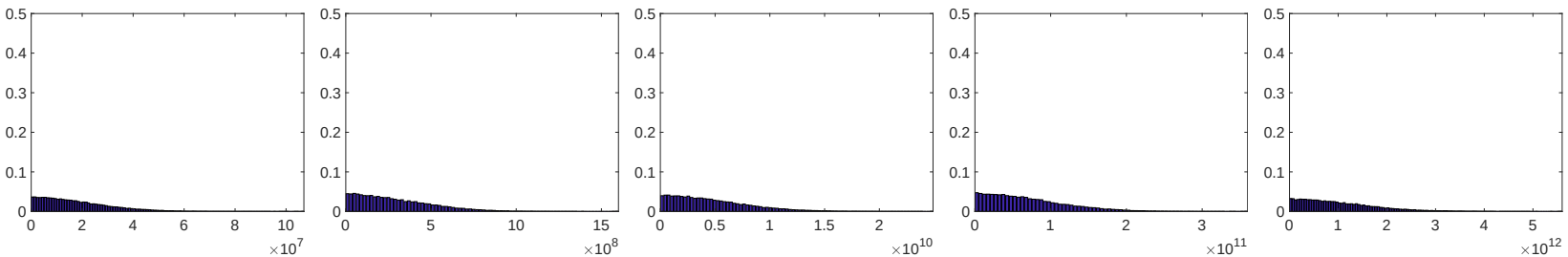
(~90-95% of activations are zero at each layer, slight artifact of the toy input)

Effect of initialization with large values on activations

- Initialize a 10-layer network with 500 nodes at each layer.
- Use a **ReLu activation function** at each layer.
- Initialize weight parameters with **large** values: $W_{i,lm} \sim N(w; 0, 1^2)$.



Layer 1 **Layer 2** **Layer 3** **Layer 4** **Layer 5**



Layer 6 **Layer 7** **Layer 8** **Layer 9** **Layer 10**

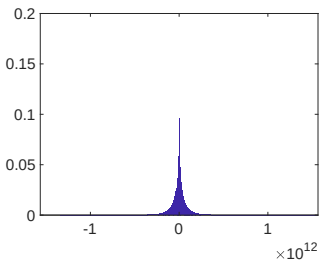
Histograms of **non-zero** activations at each layer.

(∼50% of activations are zero at each layer)

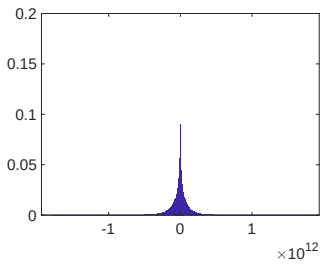
Note: values of x-axis rapidly increase with each layer.

Effect of initialization with large values on gradients

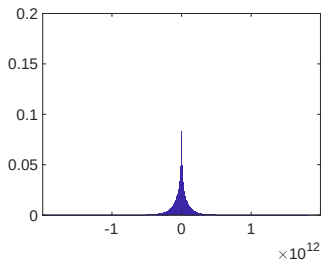
- **Set-up**: $W_{i,lm} \sim N(w; 0, 1^2)$, ReLu activation.
- Activations at each layer are increasingly large.
- What are the gradient values computed by back-prop algorithm?



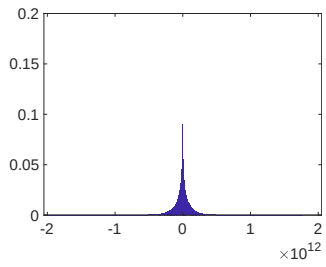
Layer 1



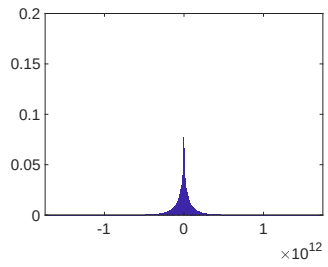
Layer 2



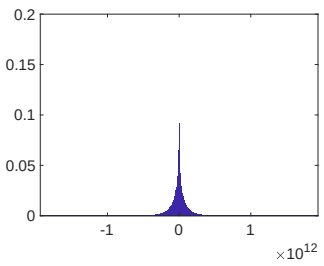
Layer 3



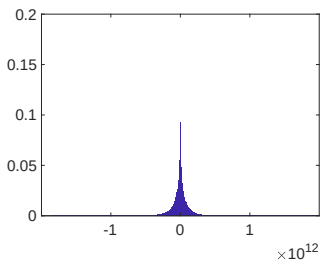
Layer 4



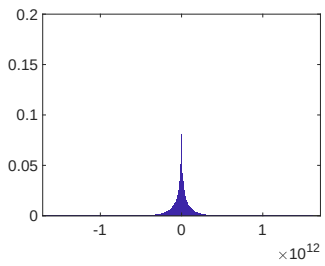
Layer 5



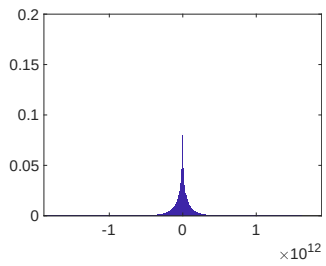
Layer 6



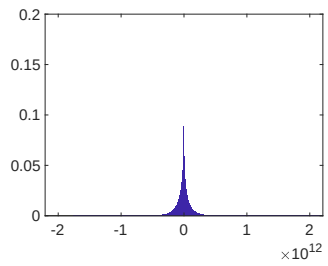
Layer 7



Layer 8



Layer 9



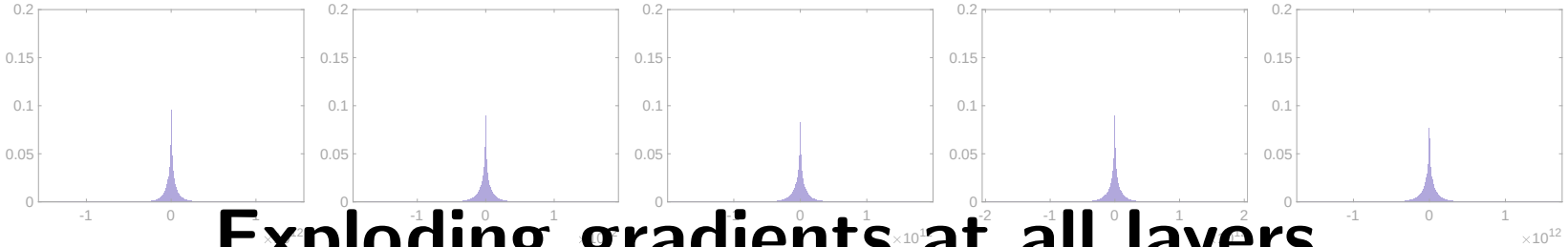
Layer 10

Histograms of **non-zero** gradients at each layer.

(~90-98% of activations are zero at each layer)

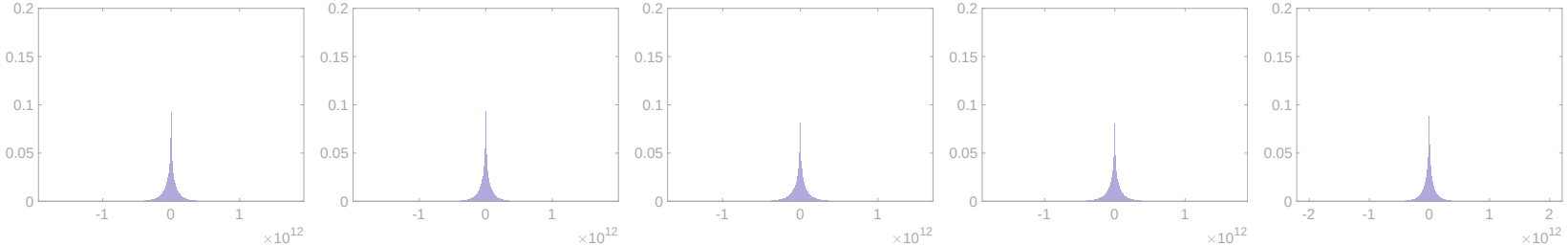
Effect of initialization with large values on gradients

- **Set-up**: $W_{i,lm} \sim N(w; 0, 1^2)$, ReLu activation.
- Activations at each layer are increasingly large.
- What are the gradient values computed by back-prop algorithm?



Exploding gradients at all layers.

Layer 1 Layer 2 Layer 3 Layer 4 Layer 5



Layer 6 Layer 7 Layer 8 Layer 9 Layer 10

Histograms of **non-zero** gradients at each layer.

(~90-98% of activations are zero at each layer)

Aggregated Backward pass for a k-layer neural network

The gradient computation for one training example $(\mathbf{x}, y)$:

- Let

$$\mathbf{g} = -(\mathbf{y} - \mathbf{p})^T$$

- for $i = k, k-1, \ldots, 1$
  1. The gradient of l w.r.t. bias vector $\mathbf{b}_i$

$$\frac{\partial l}{\partial \mathbf{b}_i} = \mathbf{g}$$

  2. Gradient of l w.r.t. weight matrix W_i

$$\frac{\partial l}{\partial W_i} = \mathbf{g}^T \mathbf{x}^{(i-1)T}$$ ← $\mathbf{x}^{(i-1)}$ contains zeros and very large numbers especially for larger i

  3. Propagate the gradient vector $\mathbf{g}$ to the previous layer (if $i > 1$)

$\mathbf{g} = \mathbf{g} W_i$ ← multiply by W_i which has many values > 1

$\mathbf{g} = \mathbf{g}\ \mathrm{diag}(\mathrm{Ind}(\mathbf{s}^{(i)} > 0))$ ← zero out some of the gradients and leave the others unchanged

Is it just a case of trial and error until we find a good value for σ?

No. For $\tanh$ activations, Xavier initialization and simple statistical reasoning to the rescue.

Xavier initialization

- **Input data**
  - Generate random input data, $N(0, 1^2)$, with $d = 500$.
- **Network architecture**
  - Initialize a 10-layer network with 500 nodes at each layer.
  - Use a $\tanh$ activation function at each layer.
- **Weight initialization with Xavier initialization**
  - Initialize weights according to:

$$W_{i,lm} \sim N(w; 0, \sigma^2) \quad \text{where } \sigma = \frac{1}{\sqrt{n_{\text{in}}}} \text{ and } W_i \text{ has size } n_{\text{out}} \times n_{\text{in}}$$

  - For our simple example $n_{\text{in}} = 500$.
- **Where does Xavier initialization come from?**
  - Want mean and spread of activations in $\mathbf{x}^{(i)}$ to be $=$ to those in $\mathbf{x}^{(i-1)}$.
  - Know $\mathbf{x}^{(i)} = \tanh(W_i \mathbf{x}^{(i-1)})$
  - If independence and simplifying assumptions are made.
  - Then $\sigma = \frac{1}{\sqrt{n_{\text{in}}}}$ pops out.

Xavier initialization

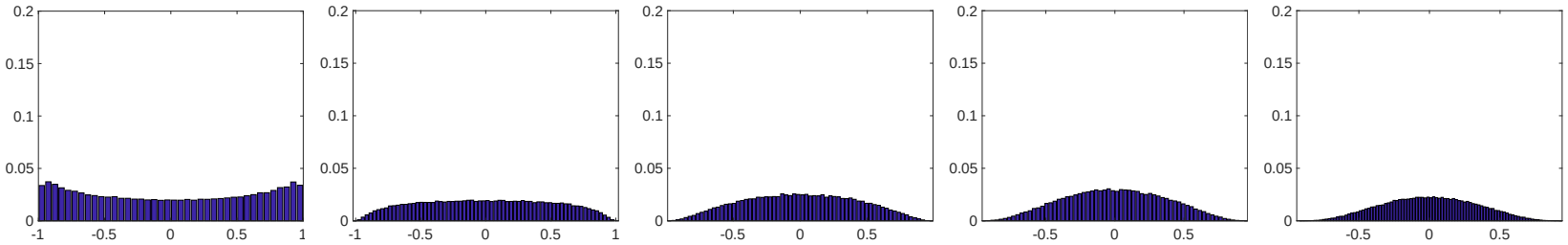
- **Input data**
  - Generate random input data, $N(0, 1^2)$, with $d = 500$.
- **Network architecture**
  - Initialize a 10-layer network with 500 nodes at each layer.
  - Use a $\tanh$ activation function at each layer.
- **Weight initialization with Xavier initialization**
  - Initialize weights according to:

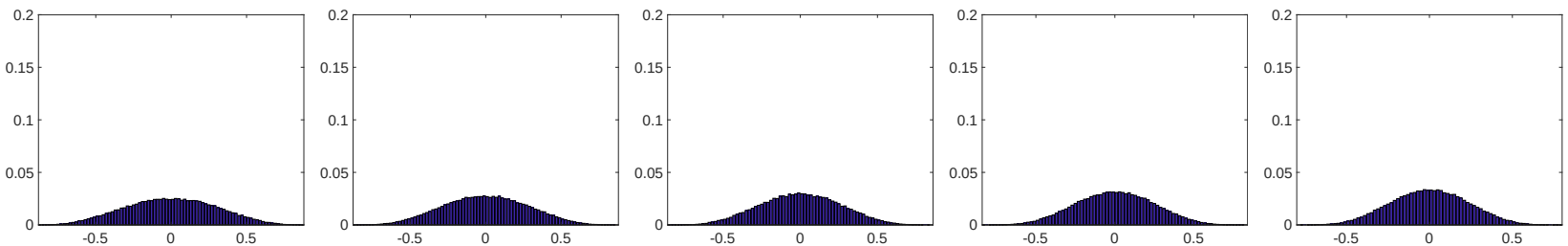
$$W_{i,lm} \sim N(w; 0, \sigma^2) \quad \text{where } \sigma = \frac{1}{\sqrt{n_{\text{in}}}} \text{ and } W_i \text{ has size } n_{\text{out}} \times n_{\text{in}}$$

  - For our simple example $n_{\text{in}} = 500$.
- **Where does Xavier initialization come from?**
  - Want mean and spread of activations in $\mathbf{x}^{(i)}$ to be $=$ to those in $\mathbf{x}^{(i-1)}$.
  - Know $\mathbf{x}^{(i)} = \tanh(W_i \mathbf{x}^{(i-1)})$
  - If independence and simplifying assumptions are made.
  - Then $\sigma = \frac{1}{\sqrt{n_{\text{in}}}}$ pops out.

Effect of Xavier initialization on distribution of activations

- Generate random input data, $N(0, 1^2)$, with $d = 500$.
- Initialize a 10-layer network with 500 nodes at each layer.
- Use a $\tanh$ **activation function** at each layer.
- Xavier initialization of weights: $W_{i,lm} \sim N(w; 0, \sigma^2)$ where $\sigma = 1/\sqrt{500}$.

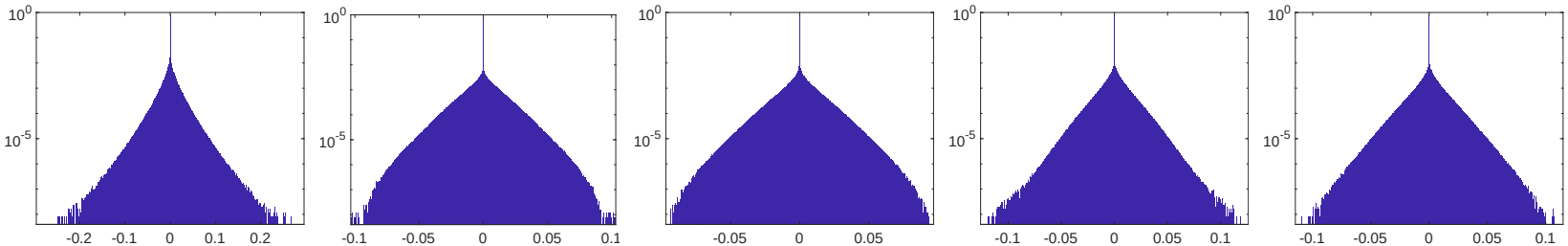


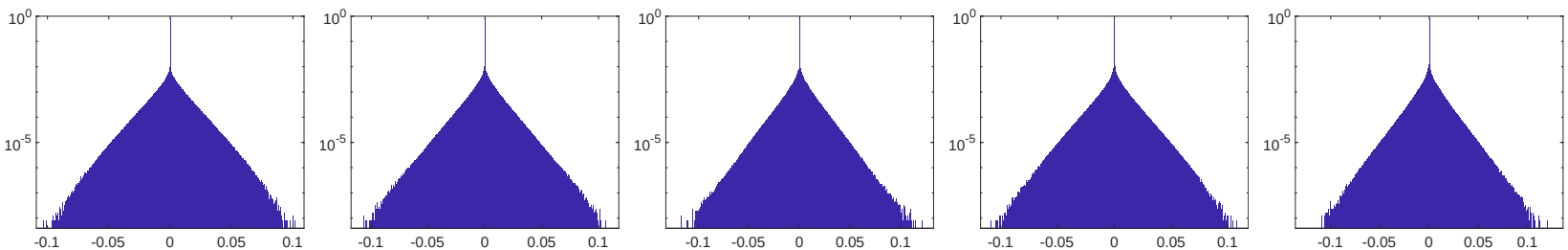


Histograms of activations at each layer

Effect on Xavier initialization on gradients

- **Set-up**: $W_{i,lm} \sim N(w; 0, 1/n_{\text{in}})$, $\tanh$ activation
- Mean and spread of activations at each layer remains $\sim$ constant.
- What are the gradient values computed by back-prop algorithm?



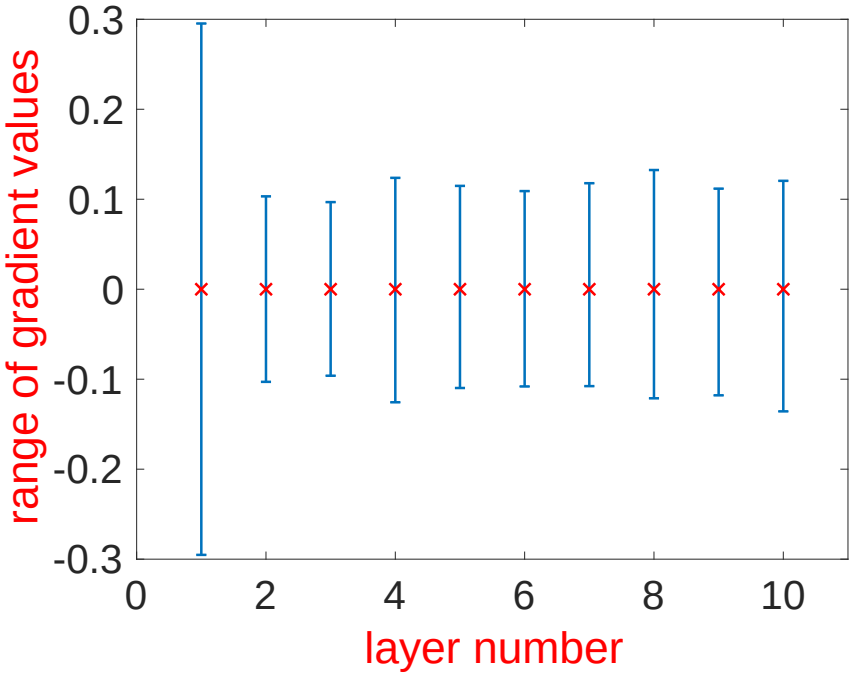


Histograms of gradients at each layer. (y-axis shown on logarithmic scale)

Note: in each graph similar range of x-values.

Effect on Xavier initialization on gradients

- **Set-up**: $W_{i,lm} \sim N(w; 0, 1/n_{\text{in}})$, $\tanh$ activation
- Mean and spread of activations at each layer remains $\sim$ constant.
- What are the gradient values computed by back-prop algorithm?



Gradient values

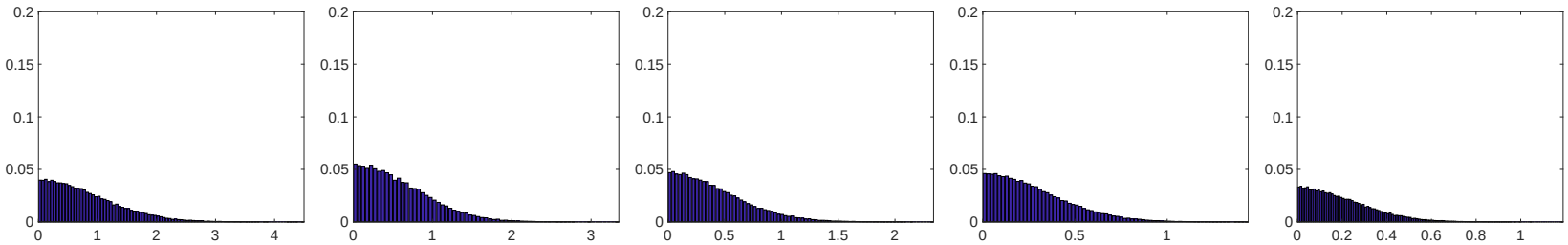
- have a nice range of values at each layer.
  (Why the increase in range for layer 1?)

Glorot initialization: $\sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}$

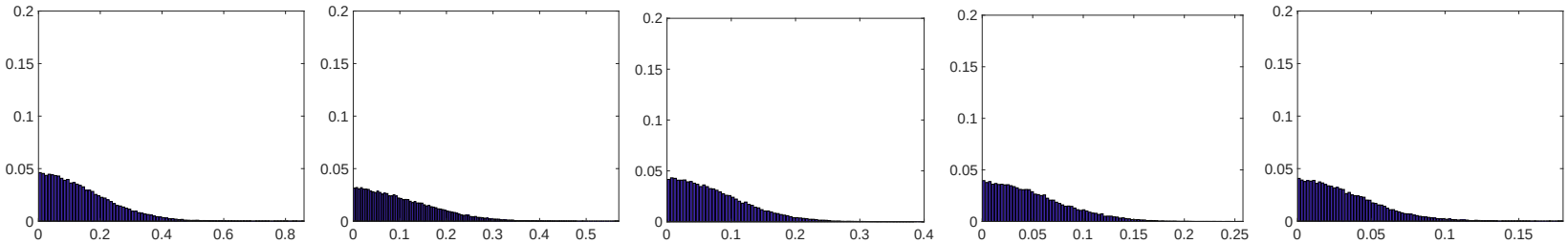
How about Xavier initialization when we have ReLu activation functions?

Xavier initialization and ReLu activation

- Generate random input data, $N(0, 1^2)$, with $d = 500$.
- Initialize a 10-layer network with 500 nodes at each layer.
- Use a **ReLu activation function** at each layer.
- Xavier initialization of weights: $W_{i,lm} \sim N(w; 0, \sigma^2)$ where $\sigma = 1/\sqrt{500}$.



Layer 1 **Layer 2** **Layer 3** **Layer 4** **Layer 5**



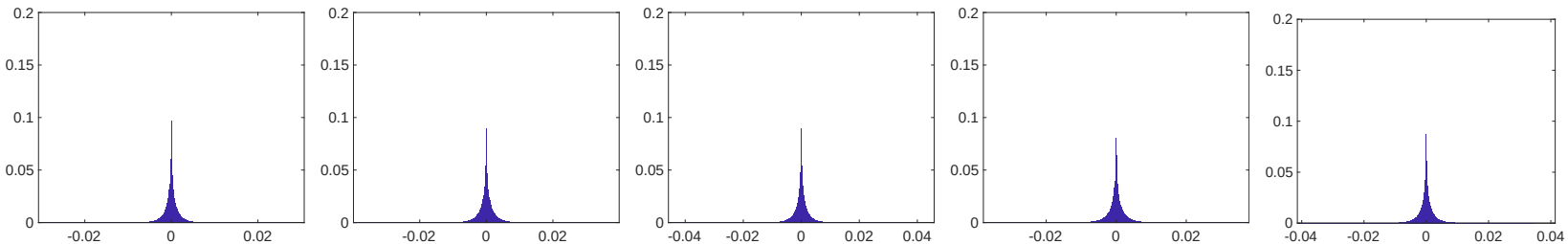
Layer 6 **Layer 7** **Layer 8** **Layer 9** **Layer 10**

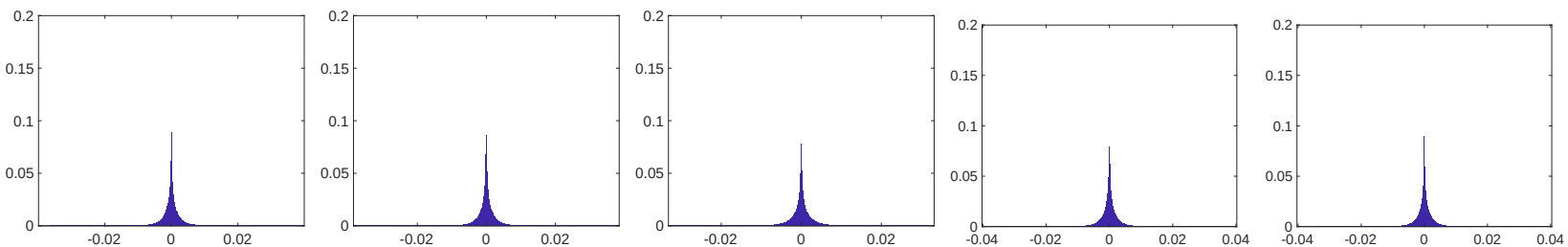
Histograms of **non-zero** activations at each layer

Note: Range of activation values get smaller with increasing layer.

Effect on Xavier initialization on gradients

- **Set-up**: $W_{i,lm} \sim N(w; 0, 1/n_{\text{in}})$, ReLu activations.
- Spread of activations at each layer slowly diminishing.
- What are the gradient values computed by back-prop algorithm?



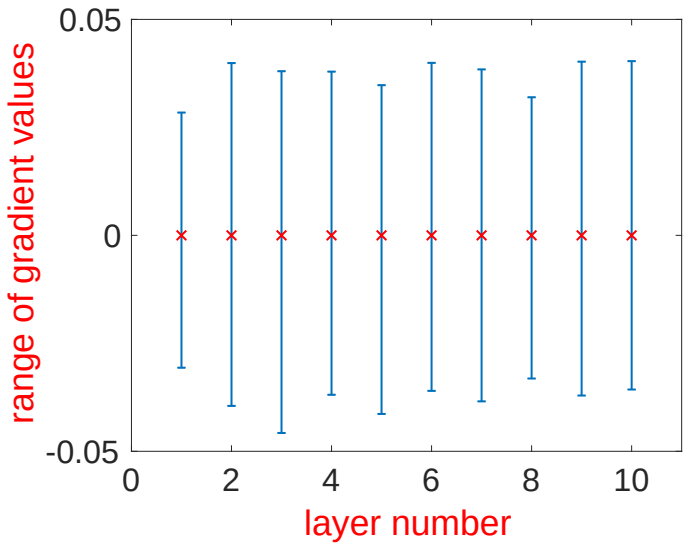


Histograms of **non-zero** gradients at each layer.

Note: in each graph similar range of x-values.

Effect on Xavier initialization on gradients

- **Set-up**: $W_{i,lm} \sim N(w; 0, 1/n_{\text{in}})$, ReLu activation
- Spread of activations at each layer slowly diminishing.
- What are the gradient values computed by back-prop algorithm?



(only non-zero gradients included in calculations)

Gradient values:

- they are small but not ridiculously small. Range remains constant across layers.

Xavier initialization diminishes effect of vanishing activations and stops exploding activations.

But can do better! **He initialization**.

He Vs *Xavier* initialization

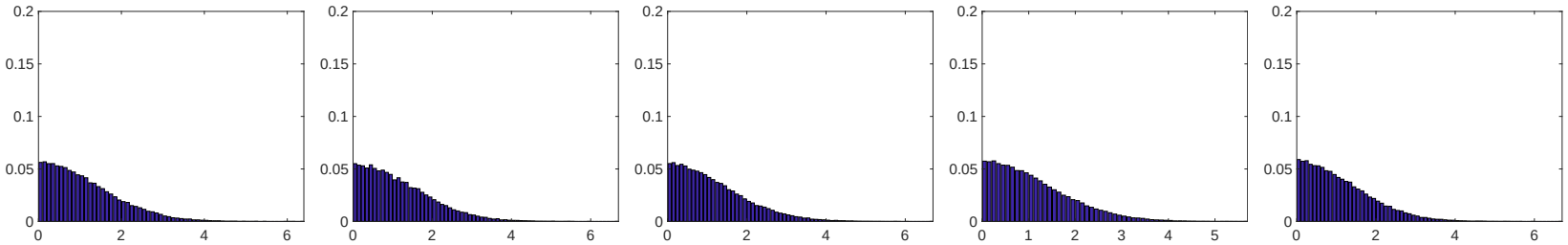
- Both *He* and *Xavier* initialization want for each layer i

$$\text{std of values in } \mathbf{x}^{(i)} = \text{std of values in } \mathbf{x}^{(i-1)}$$

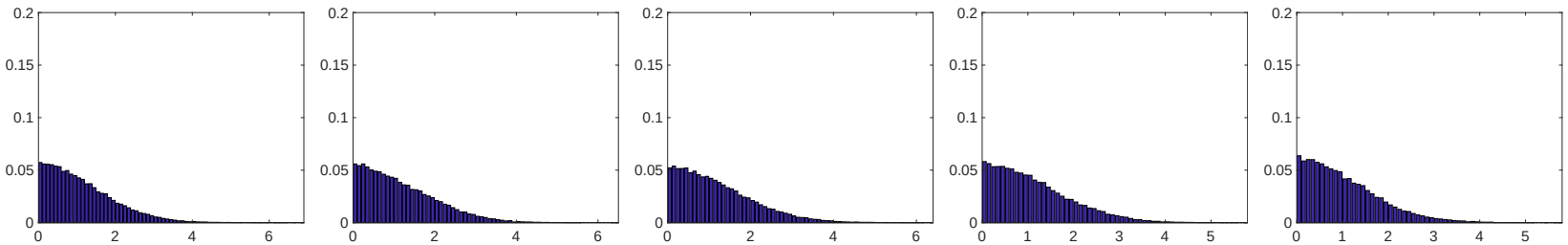
- Two approaches assume $W_{i,lm} \sim N(w; 0, \sigma^2)$ and make similar independence assumptions
- Xavier initialization assumes activations, $\mathbf{x}^{(i)}$, at layer follow a symmetric distribution.
- This assumption **not true** for networks with ReLu fns.
- *He* initialization takes this into account.
- Thus $\sigma = \sqrt{\frac{2}{n_{\text{in}}}}$ pops out.

Effect of He initialization on activations

- Generate random input data, $N(0, 1^2)$, with $d = 500$.
- Initialize a 10-layer network with 500 nodes at each layer.
- Use **ReLu activation function** at each layer.
- He initialization of weights: $W_{i,lm} \sim N(w; 0, \sigma^2)$ where $\sigma = \sqrt{2/500}$.



Layer 1 **Layer 2** **Layer 3** **Layer 4** **Layer 5**

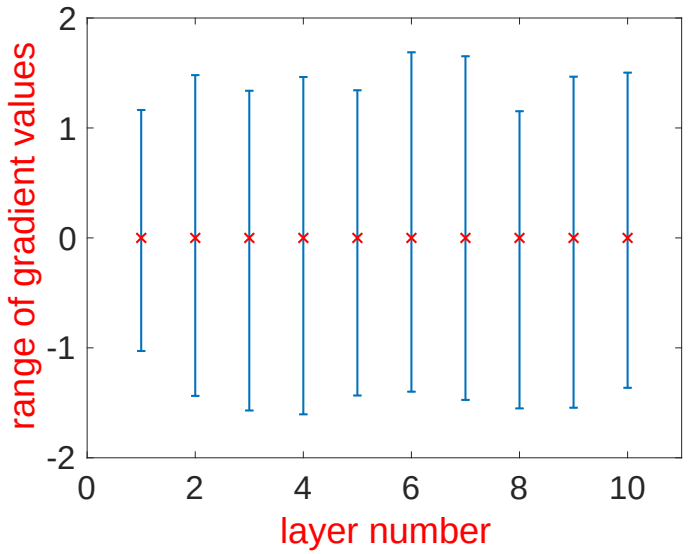


Layer 6 **Layer 7** **Layer 8** **Layer 9** **Layer 10**

Histograms of **non-zero** activations at each layer

Effect on He initialization on gradients

- **Set-up**: $W_{i,lm} \sim N(w; 0, 2/n_{\text{in}})$, ReLu activation
- Mean and spread of activations at each layer remains $\sim$ constant.
- What are the gradient values computed by back-prop algorithm?



(only non-zero gradients included in calculations)

Gradient values

- have a nice range of values at each layer

Proper Initialization an active area of research

- **Understanding the difficulty of training deep feedforward neural networks** by Glorot and Bengio, 2010
- **Exact solutions to the nonlinear dynamics of learning in deep linear neural networks** by Saxe et al, 2013
- **Random walk initialization for training very deep feedforward networks** by Sussillo and Abbott, 2014
- **Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification** by He et al., 2015
- **Data-dependent Initializations of Convolutional Neural Networks** by Krähenbühl et al., 2015
- **All you need is a good init**, Mishkin and Matas, 2015

Lessening the effect of initialization: Batch normalization

Batch Normalization

- Want unit Gaussian activations at each layer?
  Just make them unit Guassian!
- Idea introduced in:

  Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, S. Ioffe, C. Szegedy, arXiv 2015.
- Consider activations at some layer for a batch: $\mathbf{s}_1^{(j)}, \mathbf{s}_2^{(j)}, \ldots, \mathbf{s}_n^{(j)}$
- To make each dimension unit gaussian, apply:

$$\hat{\mathbf{s}}_i^{(j)} = \text{diag}(\sigma_1, \ldots, \sigma_m)^{-1} \left(\mathbf{s}_i^{(j)} - \boldsymbol{\mu} \right)$$

where

$$\boldsymbol{\mu} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{s}_i^{(j)}, \quad \sigma_p^2 = \frac{1}{n} \sum_{i=1}^{n} (s_{i,p}^{(j)} - \mu_p)^2$$

Batch Normalization

- Usually apply **normalization** after the fully connected layer before non-linearity.
- Therefore for a $k-$layer network have
  - for $i = 1, \ldots, k-1$

    for each $(\mathbf{x}^{(i-1)}, y) \in \mathcal{D}$ ← Apply ith linear transformation to batch

    $$\mathbf{s}^{(i)} = W_i \mathbf{x}^{(i-1)} + \mathbf{b}_i$$

    end

    Compute batch mean and variances of ith layer:

    $$\boldsymbol{\mu} = \frac{1}{|\mathcal{D}|} \sum_{\mathbf{s}^{(i)} \in \mathcal{D}} \mathbf{s}^{(i)}, \quad \sigma_j^2 = \frac{1}{|\mathcal{D}|} \sum_{\mathbf{s}^{(i)} \in \mathcal{D}} \left(s_j^{(i)} - \mu_j \right)^2 \text{ for } j = 1, \ldots, m_i$$

    for each $(\mathbf{s}^{(i)}, y) \in \mathcal{D}$ ← Apply BN and activation function

    $$\hat{\mathbf{s}}^{(i)} = \text{BatchNormalise}(\mathbf{s}^{(i)}, \boldsymbol{\mu}, \sigma_1, \ldots, \sigma_{m_i})$$
    $$\mathbf{x}^{(i)} = \max\left(0, \hat{\mathbf{s}}^{(i)}\right)$$

    end

  end
  - Apply final linear transformation: $\mathbf{s}^{(k)} = W_k \mathbf{x}^{(k-1)} + \mathbf{b}_k$

Batch Normalization: Scale & shift range

- Also allow the network to squash and shift the range

$$\tilde{\mathbf{s}}^{(i)} = \boldsymbol{\gamma}^{(i)} \odot \hat{\mathbf{s}}^{(i)} + \boldsymbol{\beta}^{(i)}$$

  of the $\hat{\mathbf{s}}^{(i)}$'s at each layer.

- Learn the $\boldsymbol{\gamma}^{(i)}$'s and $\boldsymbol{\beta}^{(i)}$'s and add them as parameters of the network.

- This transformation is necessary to maintain the expressiveness of the network function.

Batch Normalization at Test Time

- At test time do not have a batch.
- Instead fixed empirical mean and variances of activations at each level are used.
- These quantities estimated during training (with running averages).

Small example of effect of BN on a 6-layer network to solve CIFAR-10

Description of network and training

- Define a fully connected network with 6 layers to solve Cifar-10.
- The number of nodes in each hidden layer: 50, 10, 10, 10, 10
- Dimensions of inputs: $32 \times 32 \times 3 = 3072$
- Number of outputs: 10
- Use $45,000$ training examples.
- Use simple mini-batch gradient descent with $n_{\text{batch}} = 100$.
- Initialization: He
- Each dimension of the training input has zero-mean and standard deviation 1.

Description of network and training

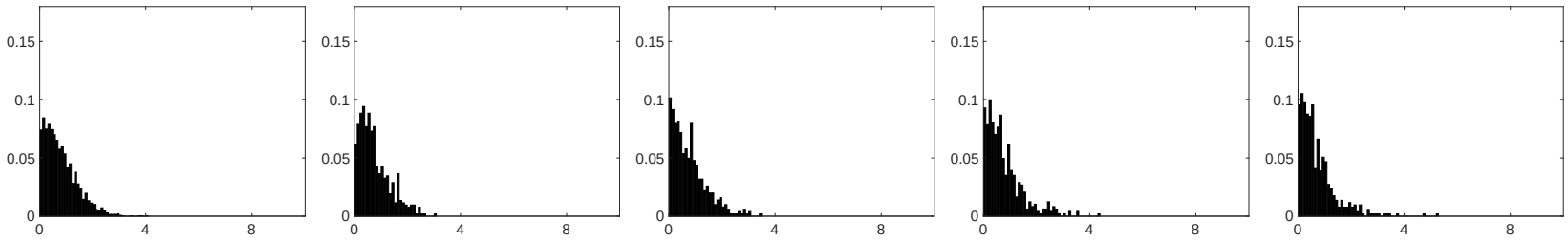
- Define a fully connected network with 6 layers to solve Cifar-10.
- The number of nodes in each hidden layer: 50, 10, 10, 10, 10
- Dimensions of inputs: $32 \times 32 \times 3 = 3072$
- Number of outputs: 10
- Use $45,000$ training examples.
- Use simple mini-batch gradient descent with $n_{\text{batch}} = 100$.
- Initialization: He
- Each dimension of the training input has zero-mean and standard deviation 1.

Investigate the effect of training with and without batch normalization on activations and gradients as η varies.

Small η: Histogram of Activations

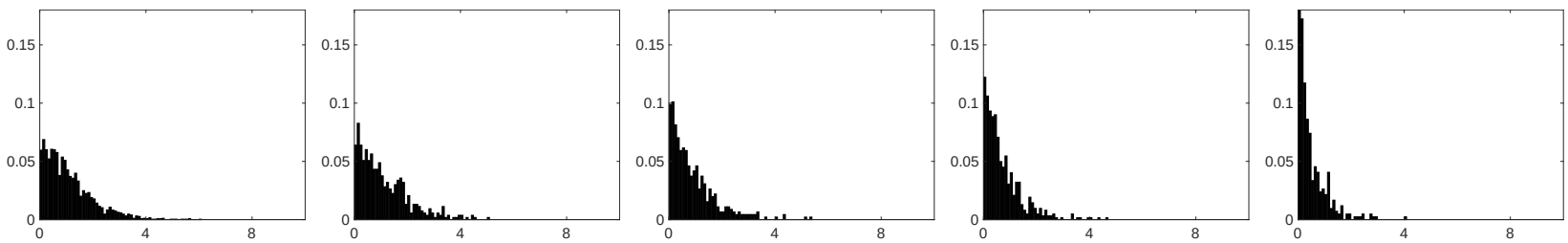
$\eta = .01$, Update step: 0

With Batch Normalization



Layer 1 **Layer 2** **Layer 3** **Layer 4** **Layer 5**

Without Batch Normalization

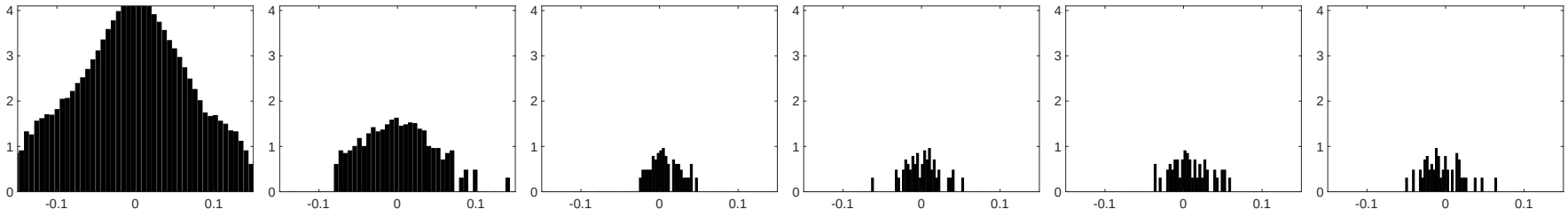


Layer 1 **Layer 2** **Layer 3** **Layer 4** **Layer 5**

Small η: Histogram of Gradients

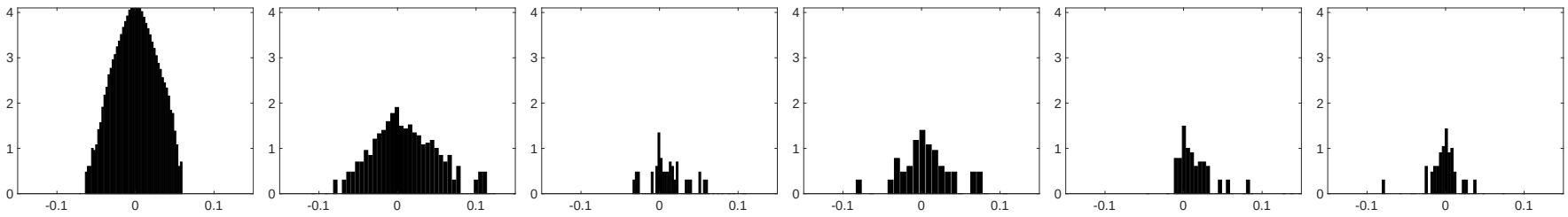
$\eta = .01$, Update step: 0

With Batch Normalization



Layer 1 **Layer 2** **Layer 3** **Layer 4** **Layer 5** **Layer 6**

Without Batch Normalization

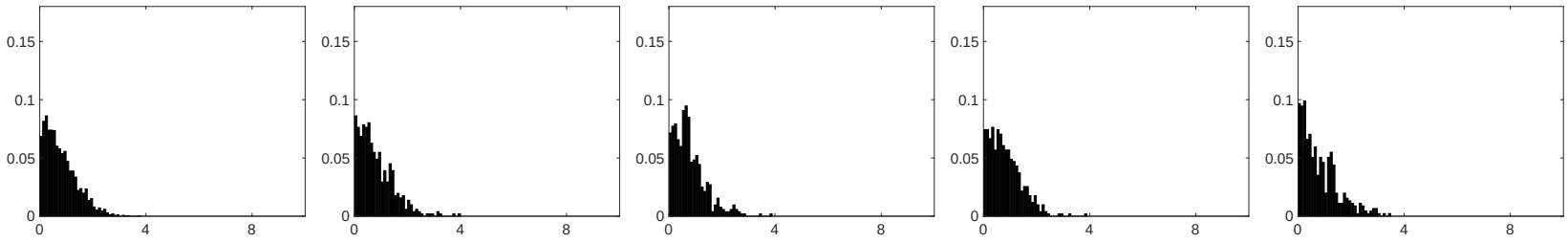


Layer 1 **Layer 2** **Layer 3** **Layer 4** **Layer 5** **Layer 6**

Small η: Histogram of Activations

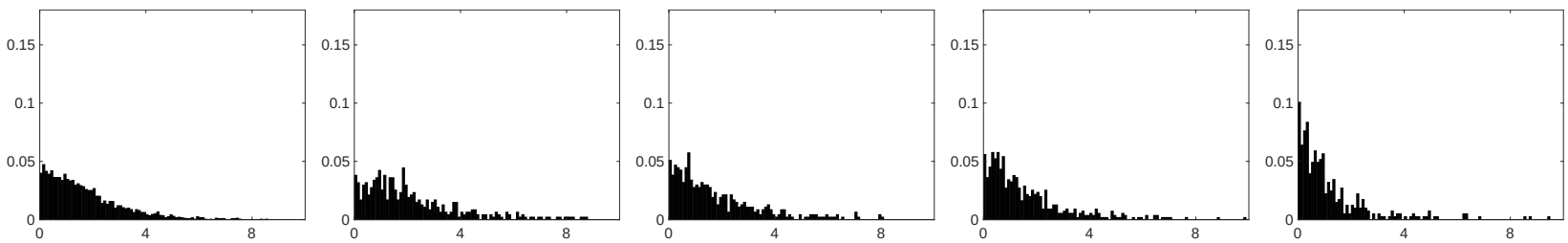
$\eta = .01$, Update step: 1000

With Batch Normalization



Layer 1 **Layer 2** **Layer 3** **Layer 4** **Layer 5**

Without Batch Normalization

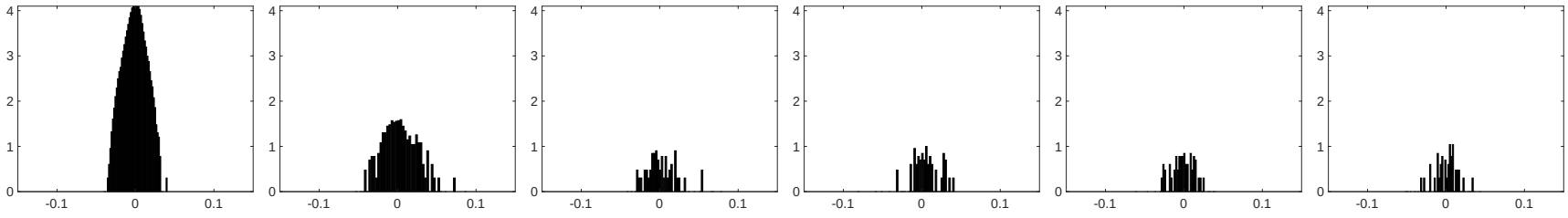


Layer 1 **Layer 2** **Layer 3** **Layer 4** **Layer 5**

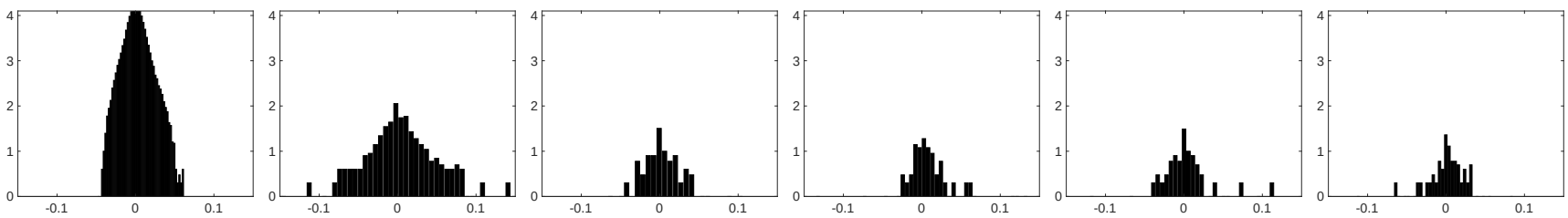
Small η: Histogram of Gradients

$\eta = .01$, Update step: 1000

With Batch Normalization



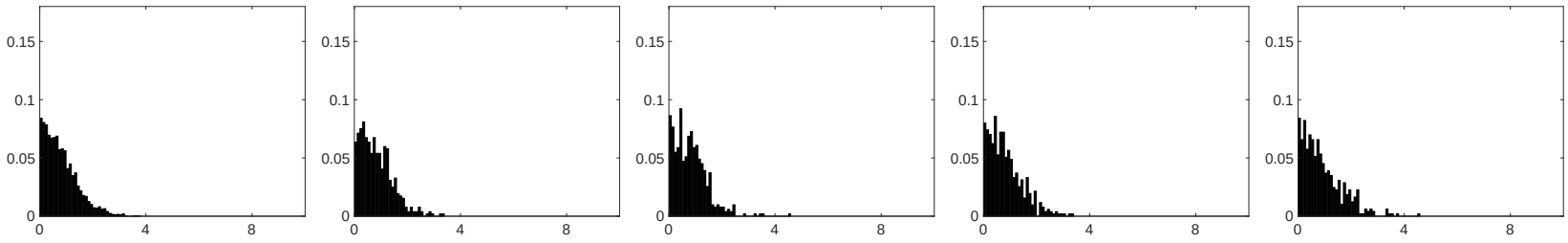
Without Batch Normalization



Small η: Histogram of Activations

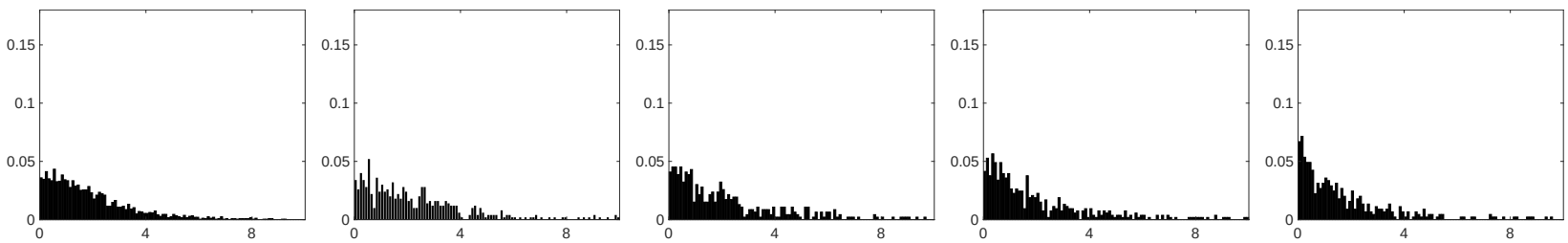
$\eta = .01$, Update step: 2000

With Batch Normalization



Layer 1 **Layer 2** **Layer 3** **Layer 4** **Layer 5**

Without Batch Normalization

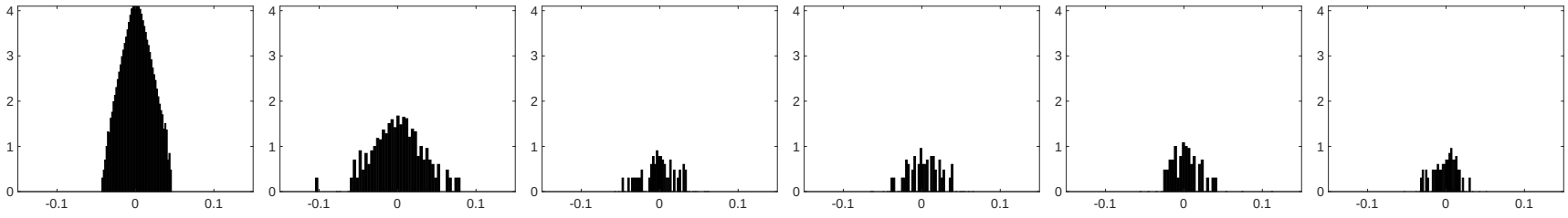


Layer 1 **Layer 2** **Layer 3** **Layer 4** **Layer 5**

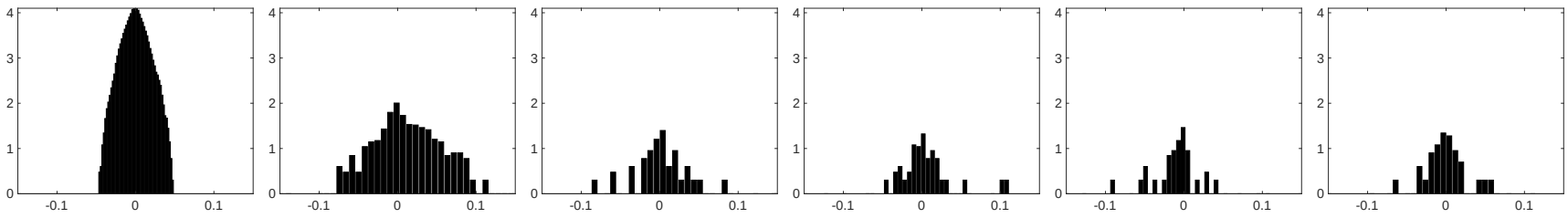
Small η: Histogram of Gradients

$\eta = .01$, Update step: 2000

With Batch Normalization



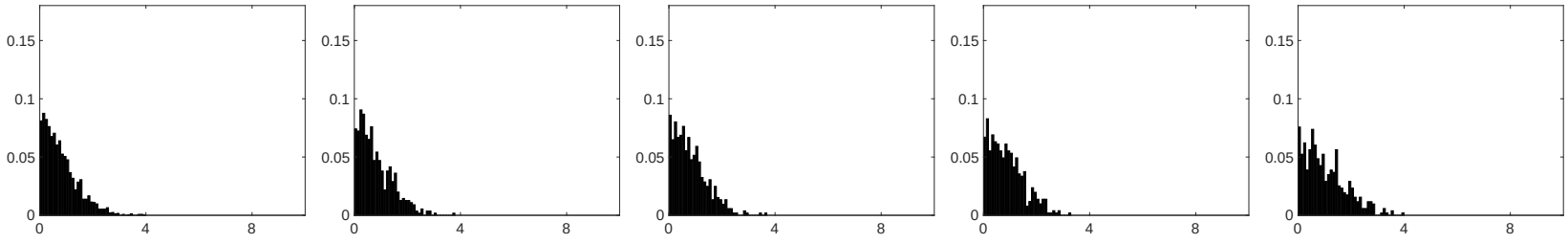
Without Batch Normalization



Histogram of Activations

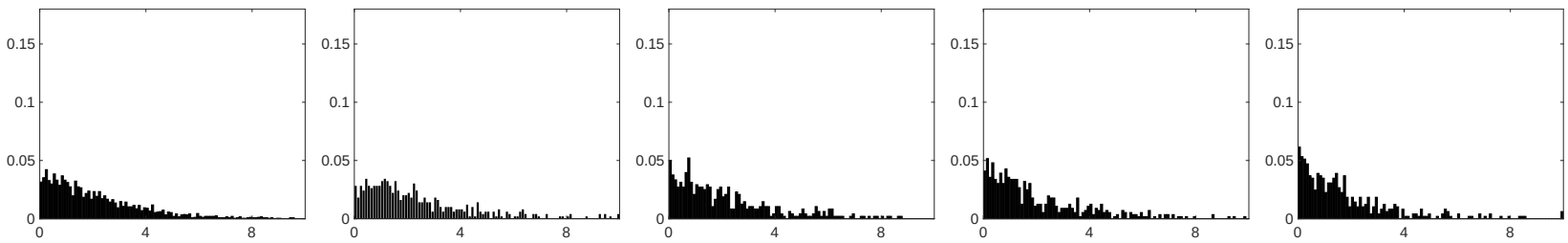
$\eta = .01$, Update step: 4000

With Batch Normalization



Layer 1 **Layer 2** **Layer 3** **Layer 4** **Layer 5**

Without Batch Normalization

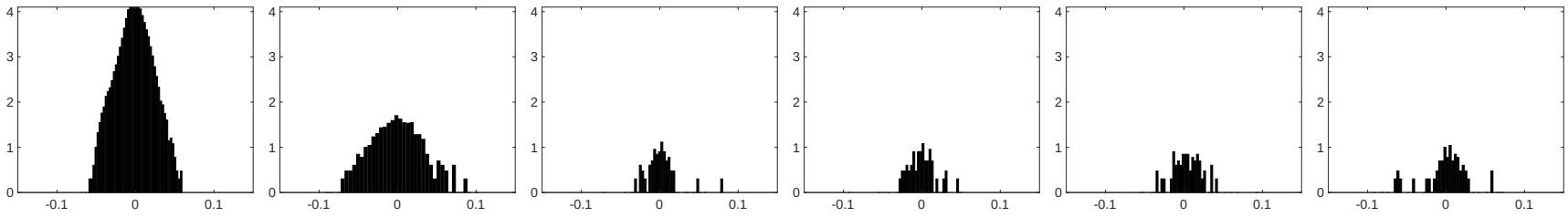


Layer 1 **Layer 2** **Layer 3** **Layer 4** **Layer 5**

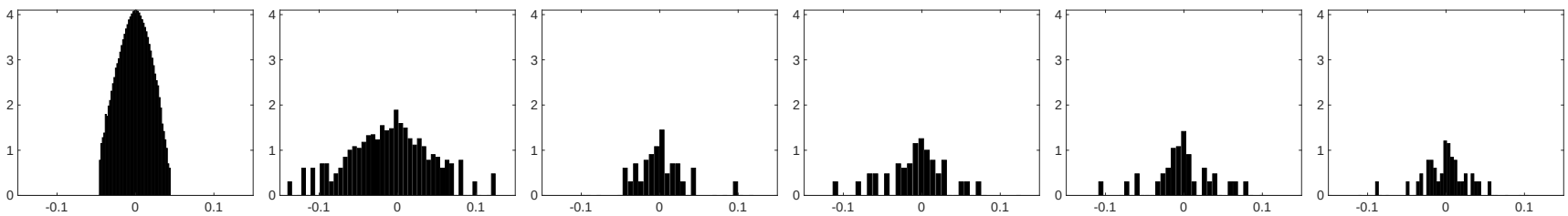
Histogram of Gradients

$\eta = .01$, Update step: 4000

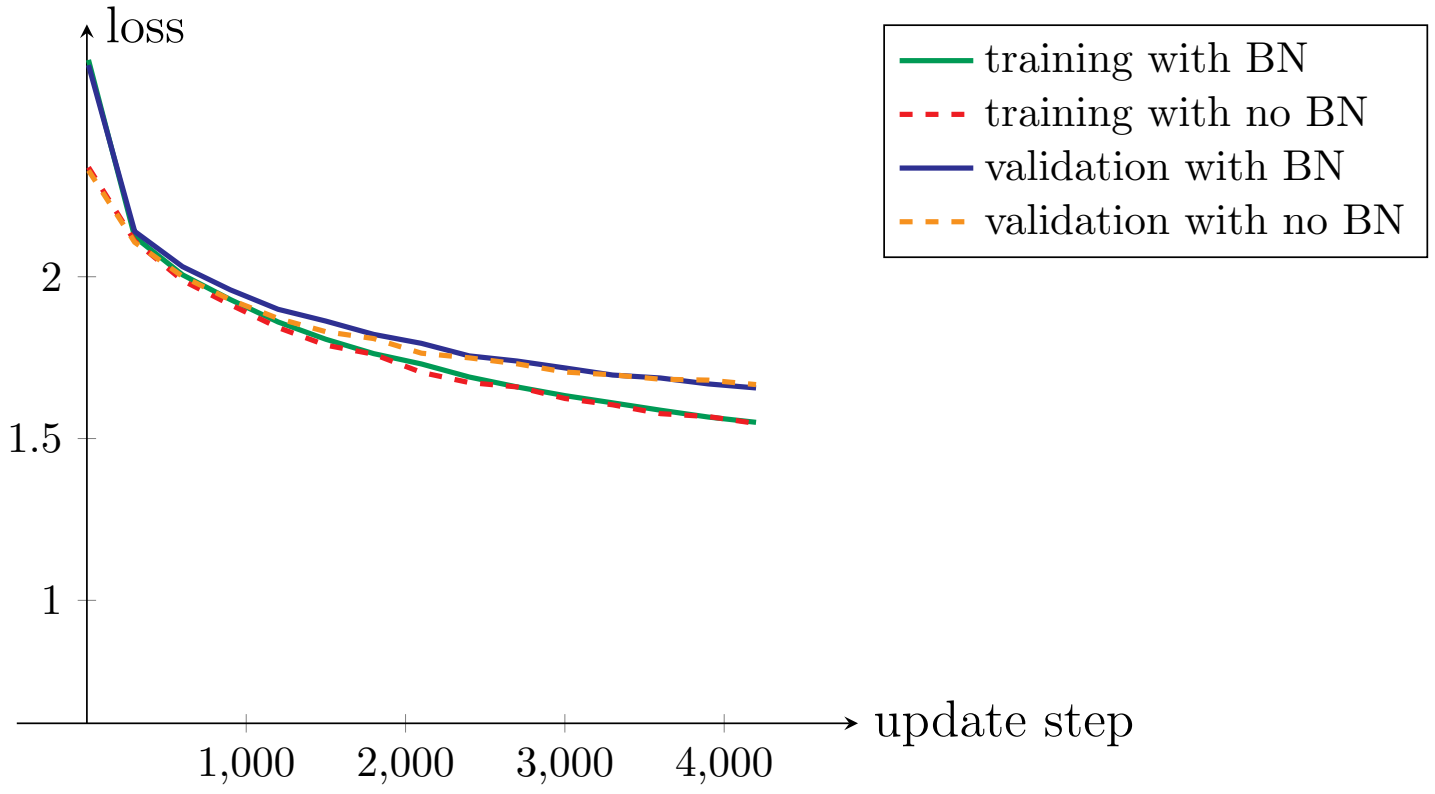
With Batch Normalization



Without Batch Normalization

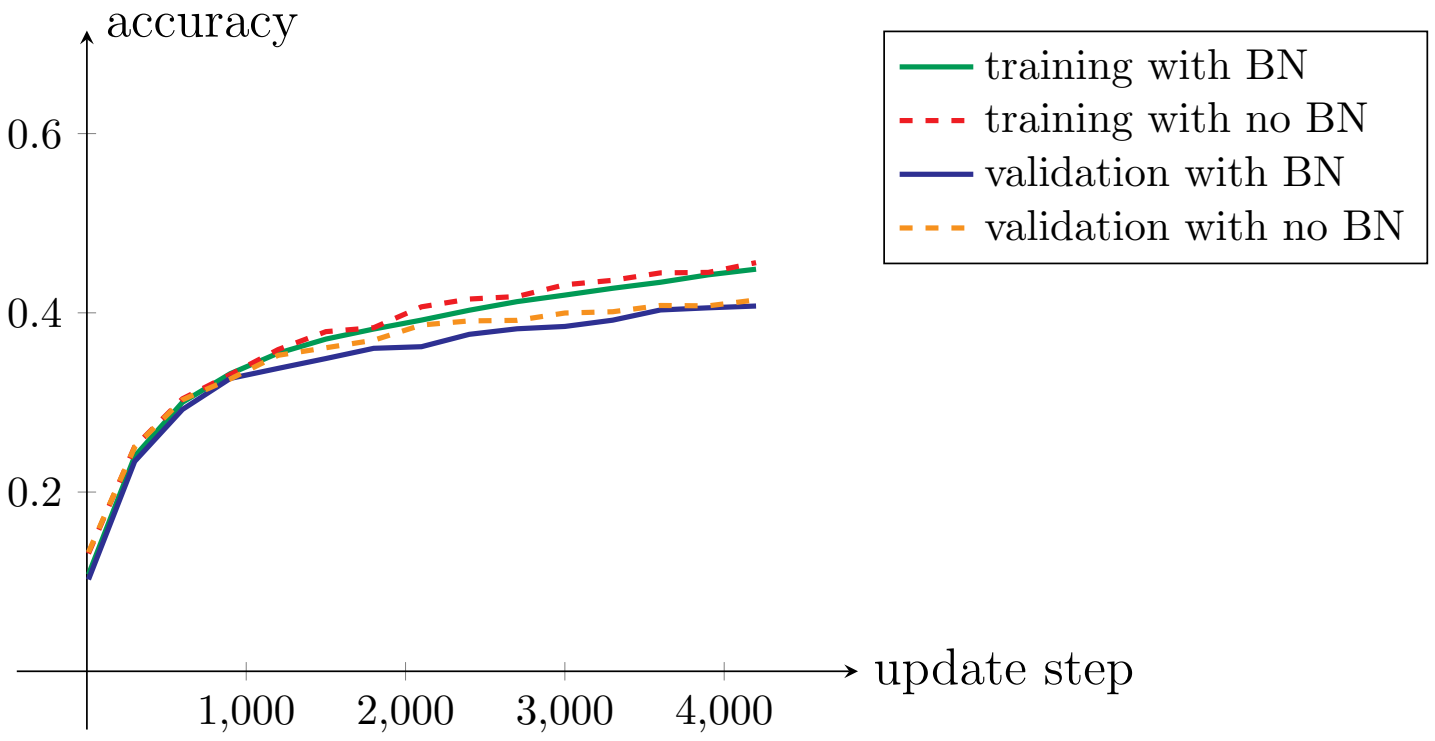


Small η: Resulting loss plots with & without BN



Loss plots with $\eta = .01$

Small η: Resulting accuracy plots with & without BN

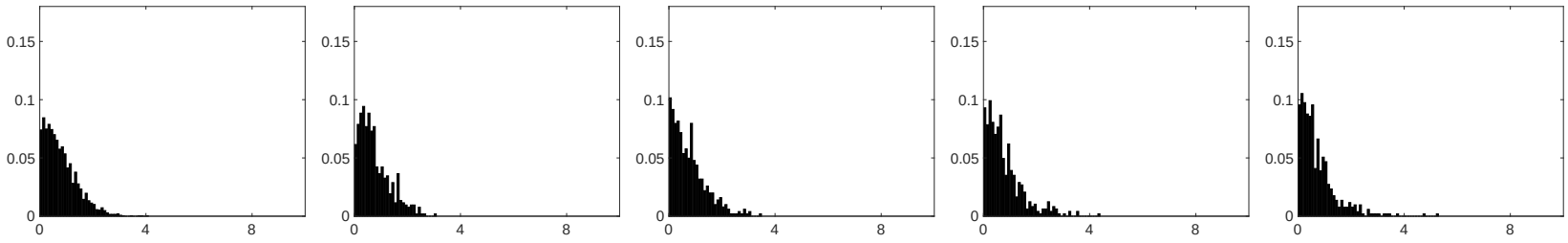


Accuracy plots with $\eta = .01$

Big η: Histogram of Activations

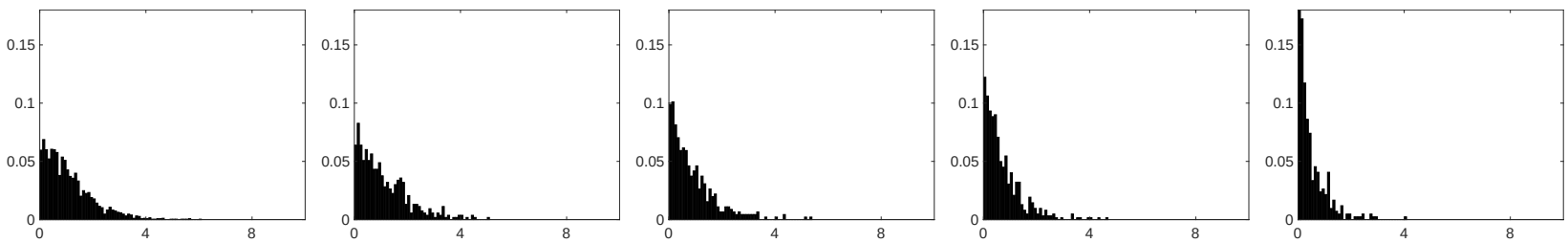
$\eta = .5$, Update step: 0

With Batch Normalization



Layer 1 **Layer 2** **Layer 3** **Layer 4** **Layer 5**

Without Batch Normalization

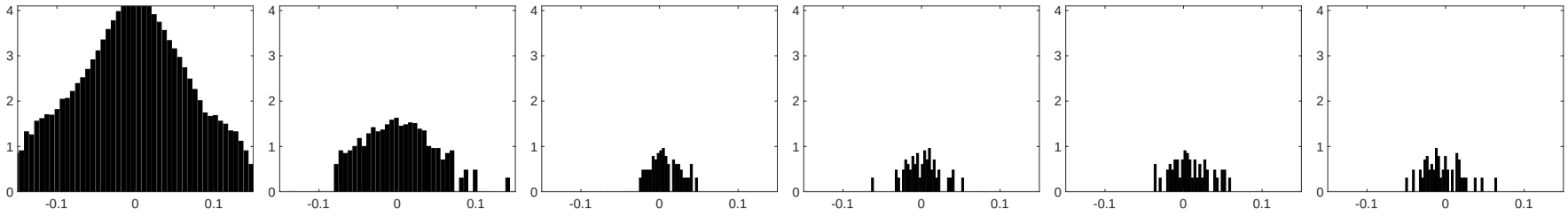


Layer 1 **Layer 2** **Layer 3** **Layer 4** **Layer 5**

Big η: Histogram of Gradients

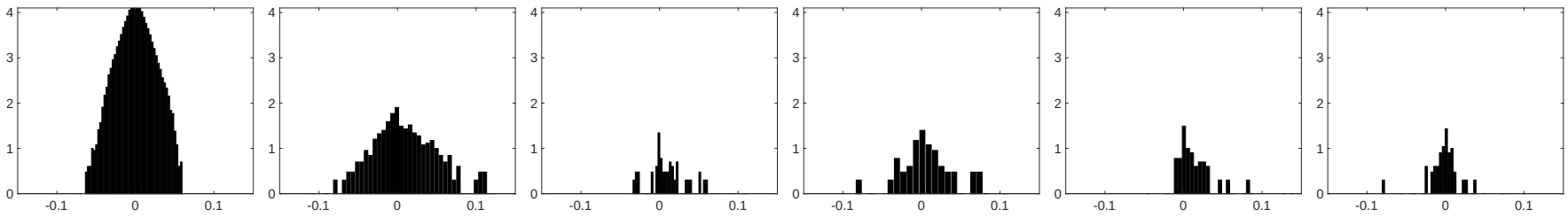
$\eta = .5$, Update step: 0

With Batch Normalization



Layer 1 **Layer 2** **Layer 3** **Layer 4** **Layer 5** **Layer 6**

Without Batch Normalization

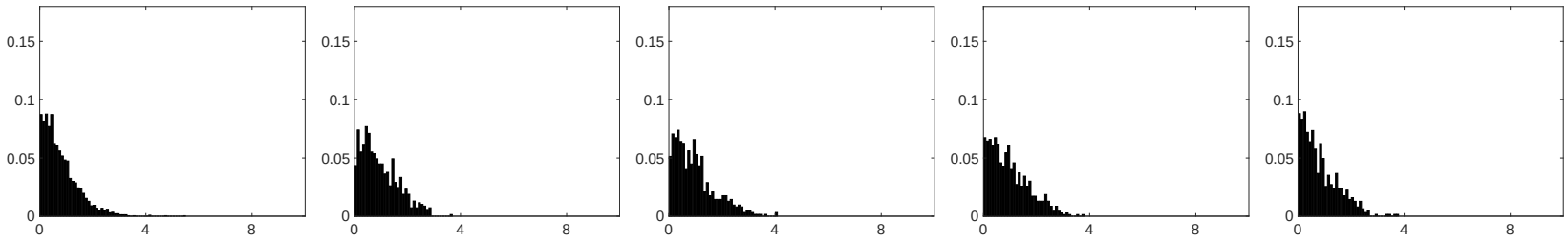


Layer 1 **Layer 2** **Layer 3** **Layer 4** **Layer 5** **Layer 6**

Big η: Histogram of Activations

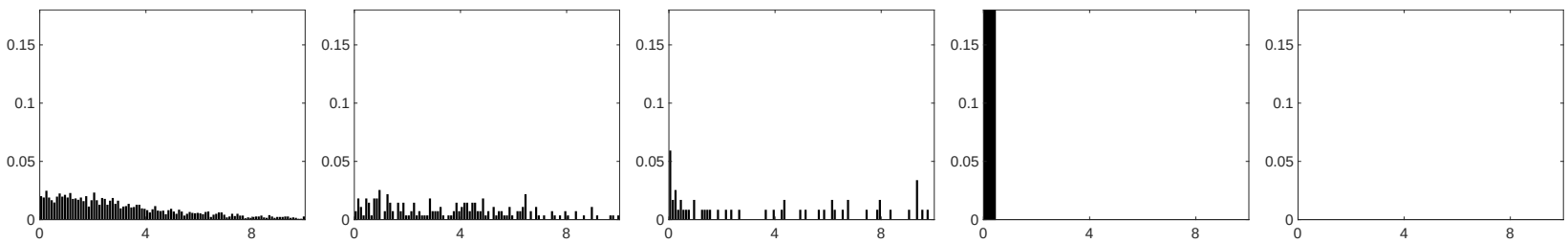
$\eta = .5$, Update step: 1000

With Batch Normalization



Layer 1 **Layer 2** **Layer 3** **Layer 4** **Layer 5**

Without Batch Normalization

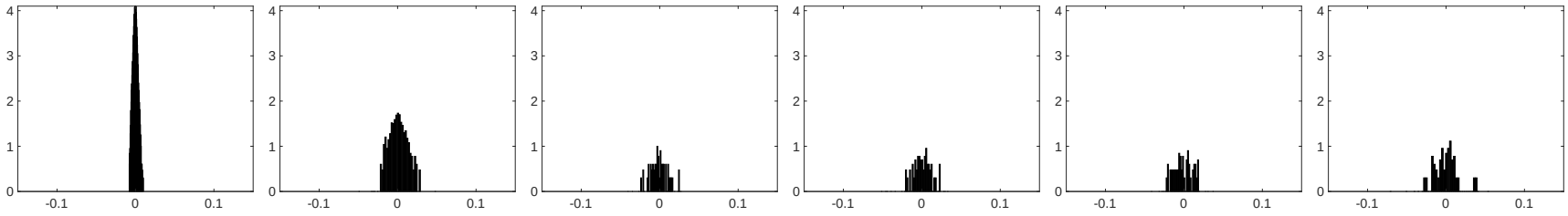


Layer 1 **Layer 2** **Layer 3** **Layer 4** **Layer 5**

Big η: Histogram of Gradients

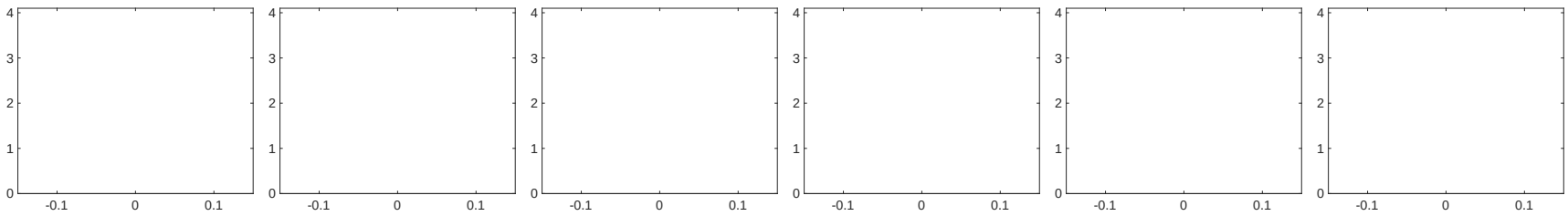
$\eta = .5$, Update step: 1000

With Batch Normalization



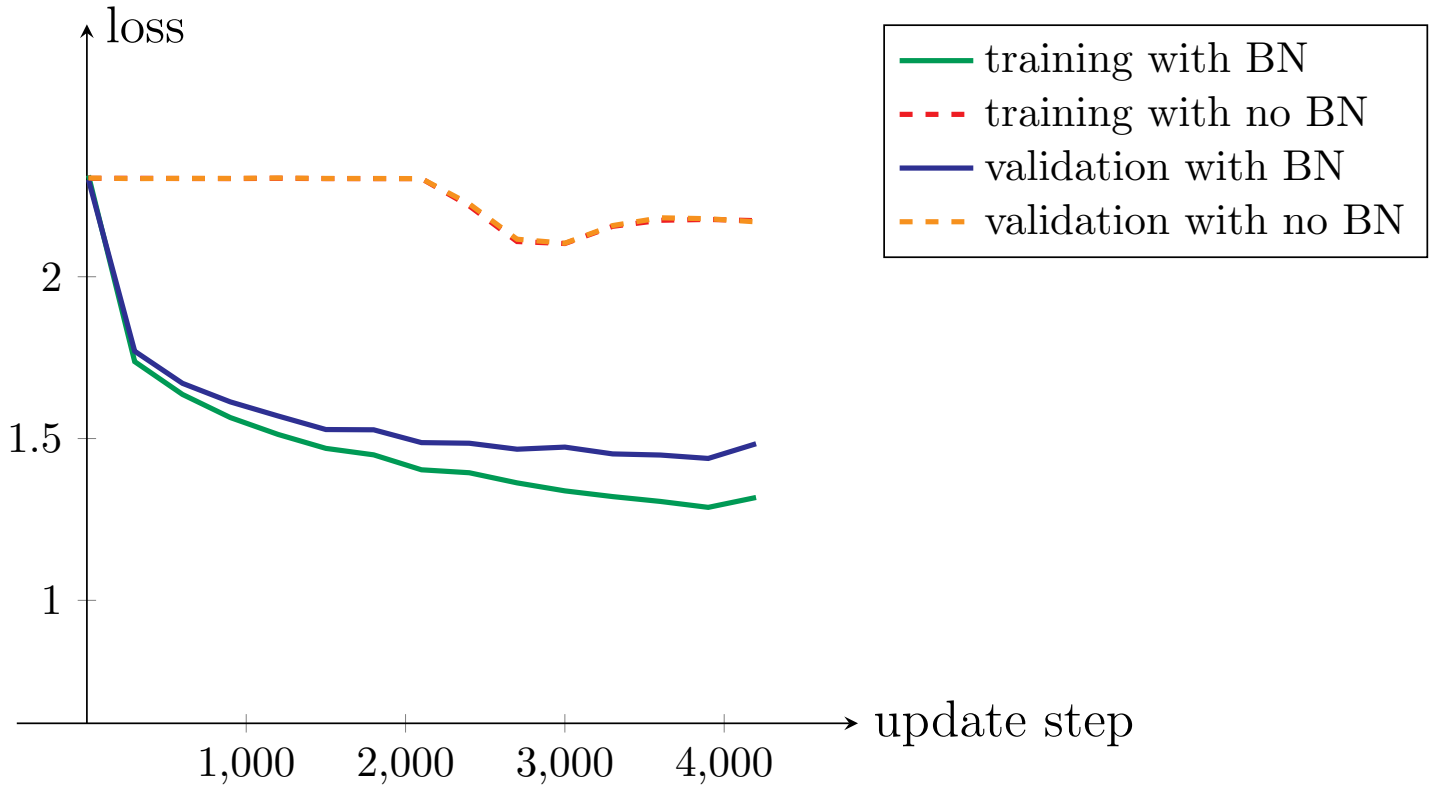
Layer 1 **Layer 2** **Layer 3** **Layer 4** **Layer 5** **Layer 6**

Without Batch Normalization



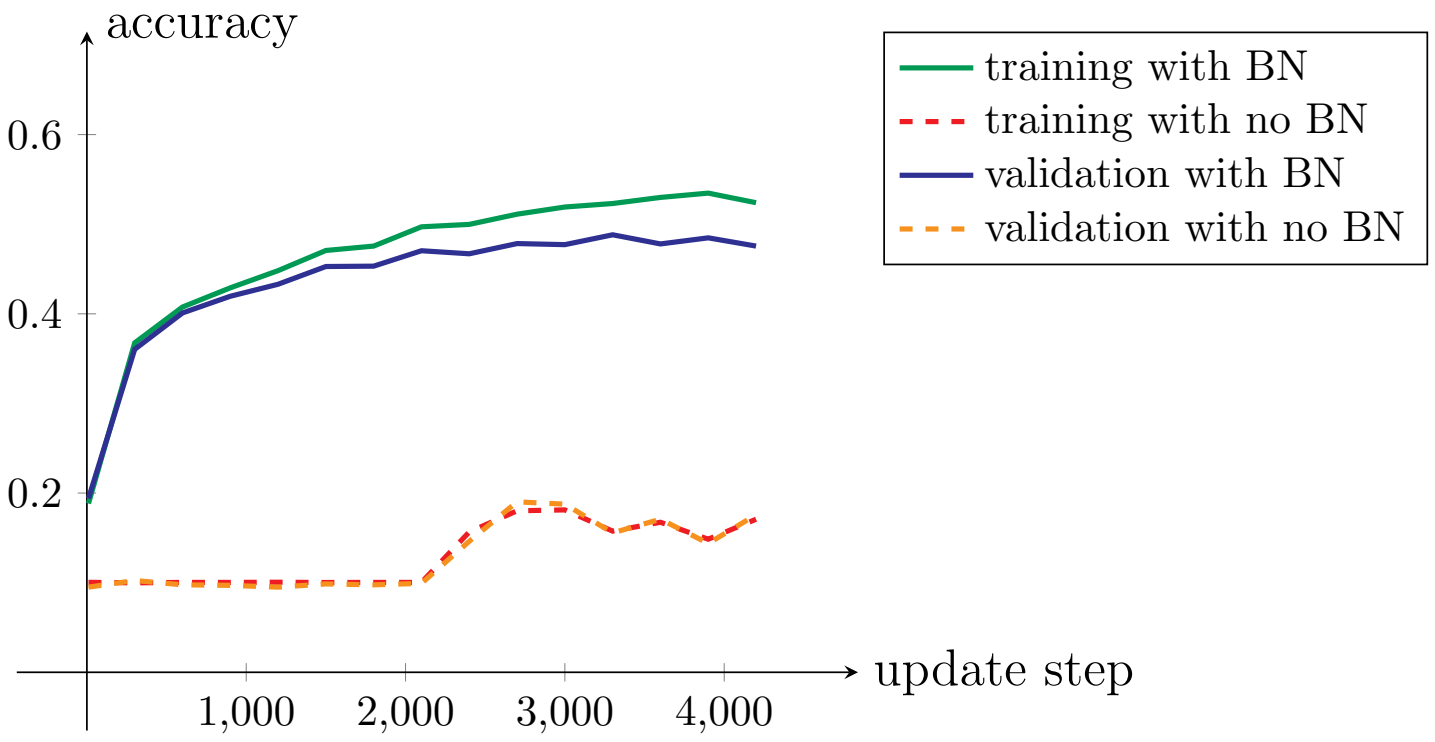
Layer 1 **Layer 2** **Layer 3** **Layer 4** **Layer 5** **Layer 6**

Big η: Resulting loss plots with & without BN



Loss plots with $\eta = .5$

Big η: Resulting accuracy plots with & without BN



Accuracy plots with $\eta = .5$

Benefits of Batch Normalization

- Makes deep networks **much** easier to train!
- Improves gradient flow through the network.
- Reduces the strong dependence on initialization.
- Allows higher learning rates $\implies$ faster convergence.
- Acts as regularization during training.
- Less stable when using small batch sizes
- Increases training time
- Can behave differently during training and testing.

Check out..

- Rethinking "Batch" in BatchNorm by Yuxin Wu and Justin Johnson

  for an exploration of good/effective practices when using Batch Normalization.

Other normalization option

Have activations at layer j: $\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_n$ with each $\mathbf{s}_i \in \mathbb{R}^d$

- **Layer Normalization**
  - Find mean and variance of entries for each example i:

$$\mu_i = \frac{1}{d}\sum_{k=1}^{d} s_{i,k} \qquad \sigma_i^2 = \frac{1}{d}\sum_{k=1}^{d}(s_{i,k} - \mu_i)^2$$

  - Then normalize each example i by its mean and std:

$$\hat{\mathbf{s}}_i = (\mathbf{s}_i - \mathbf{1}\mu_i)\,\mathrm{diag}\left(\mathbf{1}_d(\sigma_i^2 + \epsilon)\right)^{-\frac{1}{2}}$$

  - Scale and shift each example i by $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ respectively:

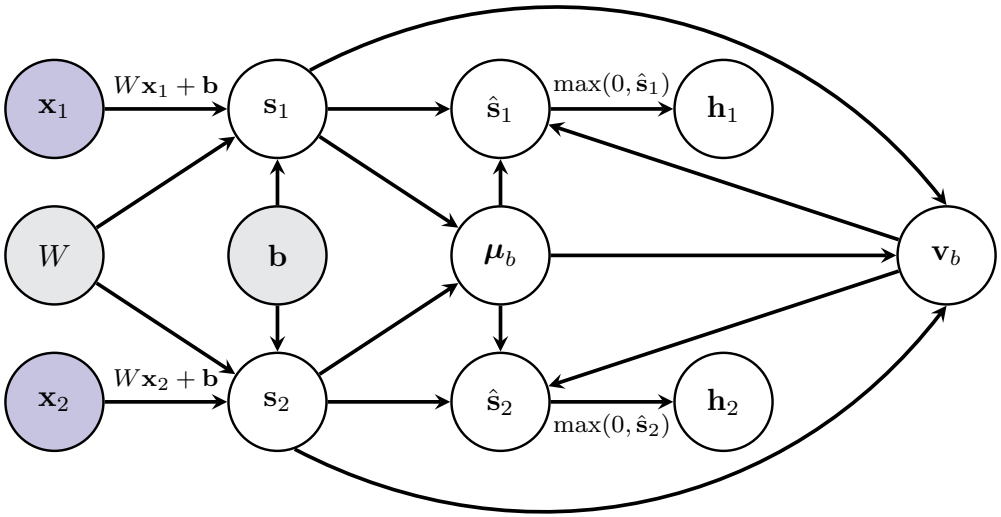
$$\tilde{\mathbf{s}}_i = \boldsymbol{\gamma} \odot \hat{\mathbf{s}}_i + \boldsymbol{\beta}$$

  - $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are learned during training.

- Same behavior at train and test!
- Used a lot in Transformer Networks.

Back-Prop for a Batch Normalization layer. (Have omitted shift and scale factors for clarity)

Computational Graph for a BN layer



- Compute the mean and variance for the *scores* in the batch:

$$\boldsymbol{\mu}_b = \frac{1}{n}\sum_{i=1}^{n} \mathbf{s}_i, \quad v_{b,j} = \frac{1}{n}\sum_{i=1}^{n} \left(s_{i,j} - \mu_{b,j}\right)^2$$

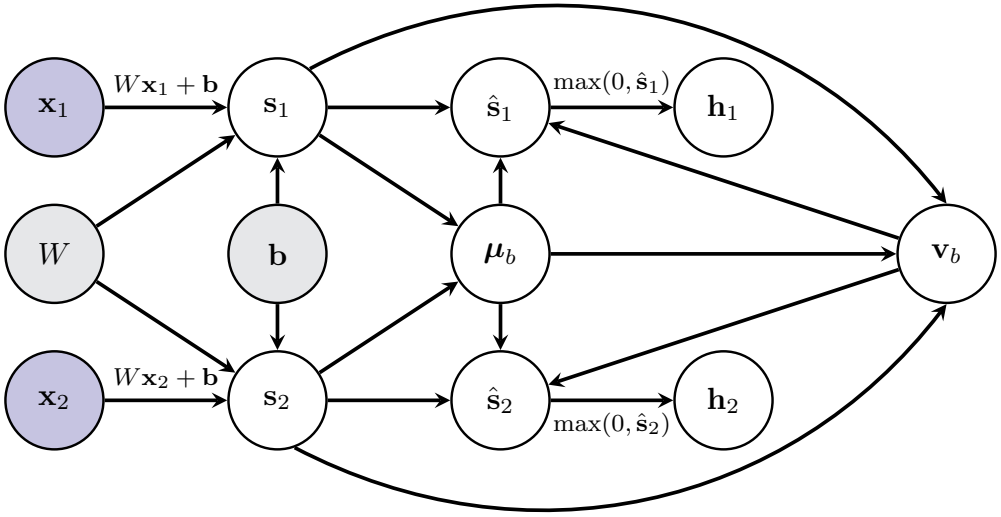
  where $\mathbf{v}_b = (v_{b,1}, v_{b,2}, \ldots, v_{b,m})^T$. ($n = 2$ in the figure.) Define

$$V_b = \text{diag}\left(\mathbf{v}_b + \epsilon\right)$$

- Apply **batch normalization** function to each *score* vector:

$$\hat{\mathbf{s}}_i = V_b^{-\frac{1}{2}}\left(\mathbf{s}_i - \boldsymbol{\mu}_b\right)$$

Gradient Computations for a BN layer

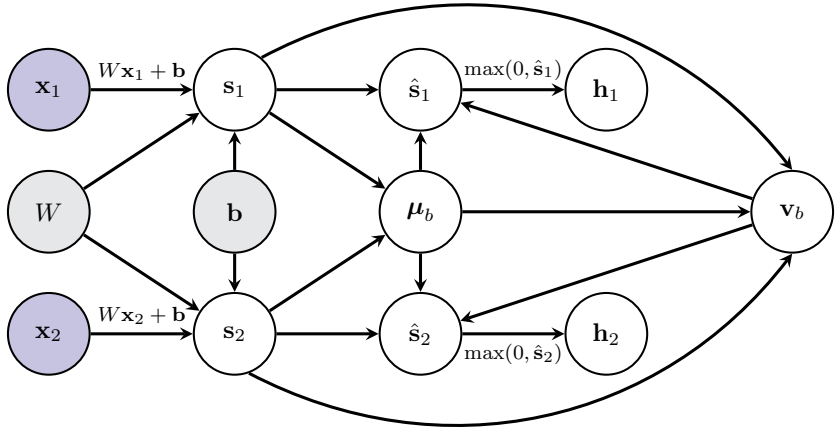


- Want to compute $\frac{\partial J}{\partial \mathbf{s}_i}$ for each $\mathbf{s}_i$ in the batch.
- The children of node $\mathbf{s}_i$ are $\{\hat{\mathbf{s}}_i, \mathbf{v}_b, \boldsymbol{\mu}_b\}$ thus

$$\frac{\partial J}{\partial \mathbf{s}_i} = \frac{\partial J}{\partial \hat{\mathbf{s}}_i}\frac{\partial \hat{\mathbf{s}}_i}{\partial \mathbf{s}_i} + \frac{\partial J}{\partial \mathbf{v}_b}\frac{\partial \mathbf{v}_b}{\partial \mathbf{s}_i} + \frac{\partial J}{\partial \boldsymbol{\mu}_b}\frac{\partial \boldsymbol{\mu}_b}{\partial \mathbf{s}_i}$$

- Let's look at the individual gradients and Jacobians.

Gradient Computations for a BN layer

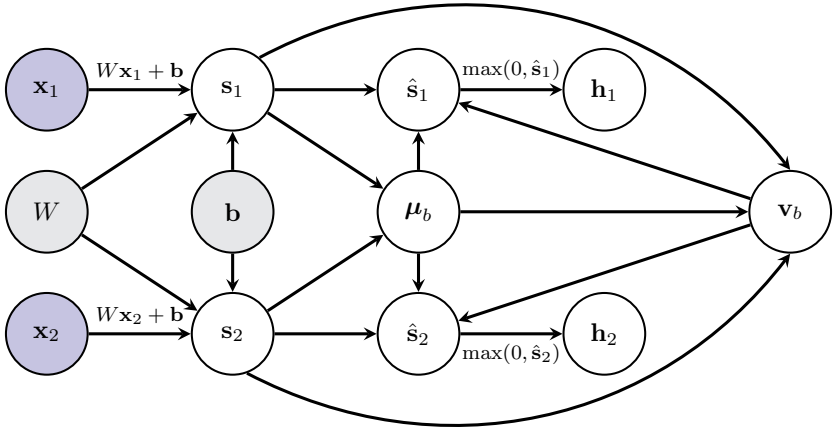


$$\frac{\partial J}{\partial \mathbf{s}_i} = \frac{\partial J}{\partial \hat{\mathbf{s}}_i} \frac{\partial \hat{\mathbf{s}}_i}{\partial \mathbf{s}_i} + \frac{\partial J}{\partial \mathbf{v}_b} \frac{\partial \mathbf{v}_b}{\partial \mathbf{s}_i} + \frac{\partial J}{\partial \boldsymbol{\mu}_b} \frac{\partial \boldsymbol{\mu}_b}{\partial \mathbf{s}_i}$$

$\uparrow$

assume already computed

Gradient Computations for a BN layer



$$\frac{\partial J}{\partial \mathbf{s}_i} = \frac{\partial J}{\partial \hat{\mathbf{s}}_i}\frac{\partial \hat{\mathbf{s}}_i}{\partial \mathbf{s}_i} + \frac{\partial J}{\partial \mathbf{v}_b}\frac{\partial \mathbf{v}_b}{\partial \mathbf{s}_i} + \frac{\partial J}{\partial \boldsymbol{\mu}_b}\frac{\partial \boldsymbol{\mu}_b}{\partial \mathbf{s}_i}$$

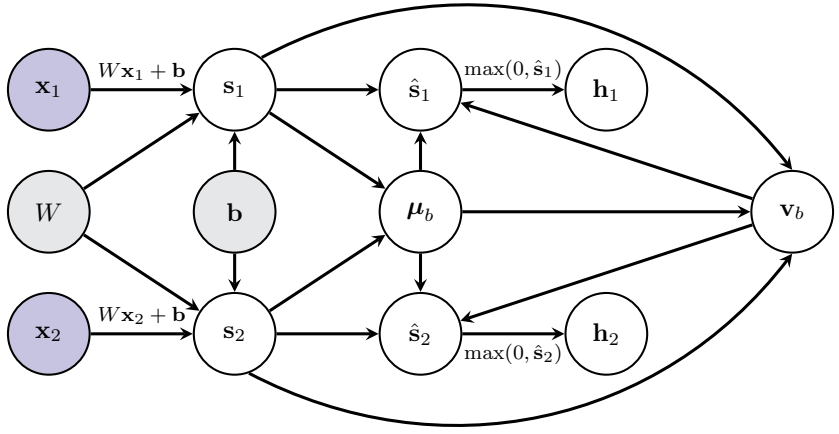
- The equation relating $\hat{\mathbf{s}}_i$ to $\mathbf{s}_i$ (remember $V_b = \text{diag}(\mathbf{v}_b + \epsilon)$)

$$\hat{\mathbf{s}}_i = V_b^{-\frac{1}{2}}(\mathbf{s}_i - \boldsymbol{\mu}_b)$$

- Therefore

$$\frac{\partial \hat{\mathbf{s}}_i}{\partial \mathbf{s}_i} = V_b^{-\frac{1}{2}}$$

Gradient Computations for a BN layer

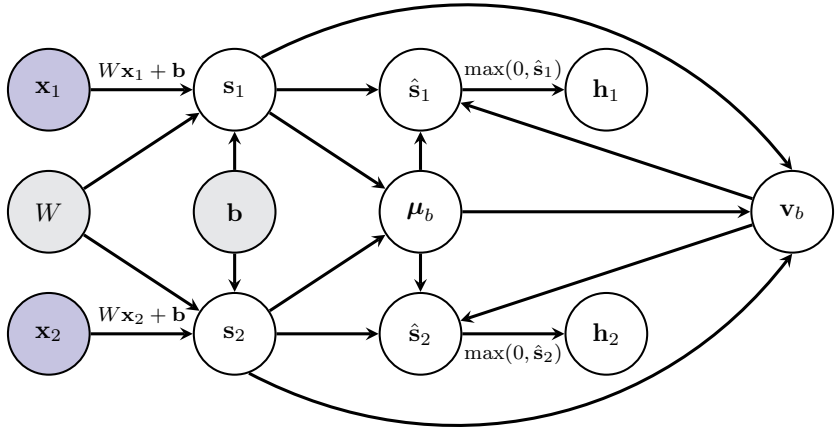


$$\frac{\partial J}{\partial \mathbf{s}_i} = \frac{\partial J}{\partial \hat{\mathbf{s}}_i}\frac{\partial \hat{\mathbf{s}}_i}{\partial \mathbf{s}_i} + \frac{\partial J}{\partial \mathbf{v}_b}\frac{\partial \mathbf{v}_b}{\partial \mathbf{s}_i} + \frac{\partial J}{\partial \boldsymbol{\mu}_b}\frac{\partial \boldsymbol{\mu}_b}{\partial \mathbf{s}_i}$$

- The children of node $\mathbf{v}_b$ are $\{\hat{\mathbf{s}}_1, \ldots, \hat{\mathbf{s}}_n\}$
- Therefore

$$\frac{\partial J}{\partial \mathbf{v}_b} = \sum_{i=1}^{n} \frac{\partial J}{\partial \hat{\mathbf{s}}_i}\frac{\partial \hat{\mathbf{s}}_i}{\partial \mathbf{v}_b}$$

Gradient Computations for a BN layer



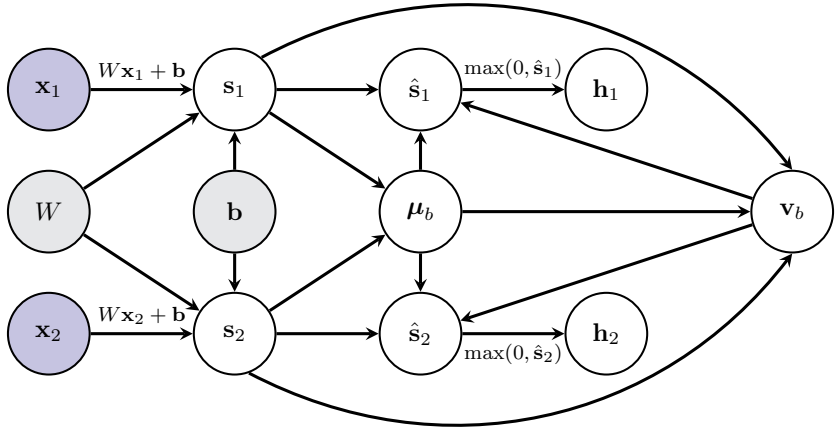
$$\frac{\partial J}{\partial \mathbf{s}_i} = \frac{\partial J}{\partial \hat{\mathbf{s}}_i}\frac{\partial \hat{\mathbf{s}}_i}{\partial \mathbf{s}_i} + \frac{\partial J}{\partial \mathbf{v}_b}\frac{\partial \mathbf{v}_b}{\partial \mathbf{s}_i} + \frac{\partial J}{\partial \boldsymbol{\mu}_b}\frac{\partial \boldsymbol{\mu}_b}{\partial \mathbf{s}_i}$$

- The children of node $\mathbf{v}_b$ are $\{\hat{\mathbf{s}}_1, \ldots, \hat{\mathbf{s}}_n\}$
- Therefore

$$\frac{\partial J}{\partial \mathbf{v}_b} = \sum_{i=1}^{n} \frac{\partial J}{\partial \hat{\mathbf{s}}_i} \frac{\partial \hat{\mathbf{s}}_i}{\partial \mathbf{v}_b}$$

$\uparrow$ assume known

Gradient Computations for a BN layer



$$\frac{\partial J}{\partial \mathbf{s}_i} = \frac{\partial J}{\partial \hat{\mathbf{s}}_i}\frac{\partial \hat{\mathbf{s}}_i}{\partial \mathbf{s}_i} + \frac{\partial J}{\partial \mathbf{v}_b}\frac{\partial \mathbf{v}_b}{\partial \mathbf{s}_i} + \frac{\partial J}{\partial \boldsymbol{\mu}_b}\frac{\partial \boldsymbol{\mu}_b}{\partial \mathbf{s}_i}$$

- The children of node $\mathbf{v}_b$ are $\{\hat{\mathbf{s}}_1, \ldots, \hat{\mathbf{s}}_n\}$
- Therefore

$$\frac{\partial J}{\partial \mathbf{v}_b} = \sum_{i=1}^{n} \frac{\partial J}{\partial \hat{\mathbf{s}}_i} \frac{\partial \hat{\mathbf{s}}_i}{\partial \mathbf{v}_b}$$

$\uparrow$ compute now

Gradient Computations for a BN layer

- The equation relating $\hat{\mathbf{s}}_i$ to $\mathbf{v}_b$ (remember $V_b = \text{diag}(\mathbf{v}_b + \epsilon)$)

$$\hat{\mathbf{s}}_i = V_b^{-\frac{1}{2}} \left(\mathbf{s}_i - \boldsymbol{\mu}_b\right)$$

- The local Jacobian we want to compute

$$\frac{\partial \hat{\mathbf{s}}_i}{\partial \mathbf{v}_b} = \begin{pmatrix} \frac{\partial \hat{s}_{i,1}}{\partial v_{b,1}} & \cdots & \frac{\partial \hat{s}_{i,1}}{\partial v_{b,m}} \\ \vdots & \vdots & \vdots \\ \frac{\partial \hat{s}_{i,m}}{\partial v_{b,1}} & \cdots & \frac{\partial \hat{s}_{i,m}}{\partial v_{b,m}} \end{pmatrix}$$

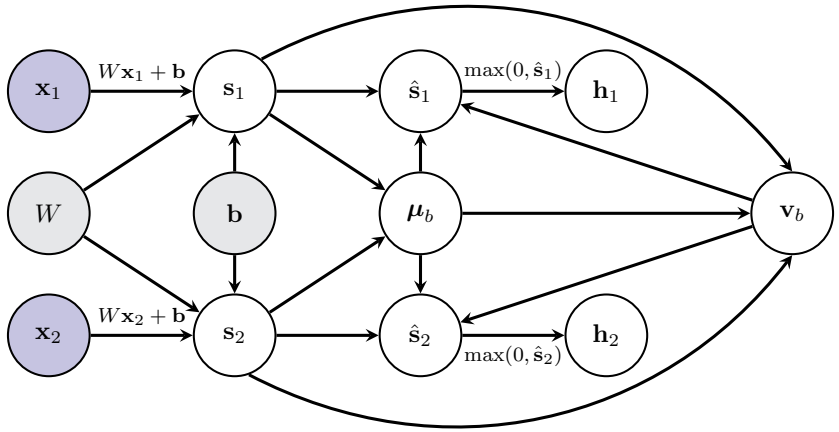
- Computing the derivative for each individual element:

$$\frac{\partial \hat{s}_{i,j}}{\partial v_{b,k}} = \begin{cases} -\frac{1}{2}(v_{b,k} + \epsilon)^{-\frac{3}{2}} \left(s_{i,k} - \mu_{b,k}\right) & \text{if } j = k \\ 0 & \text{otherwise} \end{cases}$$

- In matrix form

$$\frac{\partial \hat{\mathbf{s}}_i}{\partial \mathbf{v}_b} = -\frac{1}{2} V_b^{-\frac{3}{2}} \,\text{diag}\left(\mathbf{s}_i - \boldsymbol{\mu}_b\right)$$

Gradient Computations for a BN layer



$$\frac{\partial J}{\partial \mathbf{s}_i} = \frac{\partial J}{\partial \hat{\mathbf{s}}_i}\frac{\partial \hat{\mathbf{s}}_i}{\partial \mathbf{s}_i} + \frac{\partial J}{\partial \mathbf{v}_b}\frac{\partial \mathbf{v}_b}{\partial \mathbf{s}_i} + \frac{\partial J}{\partial \boldsymbol{\mu}_b}\frac{\partial \boldsymbol{\mu}_b}{\partial \mathbf{s}_i}$$

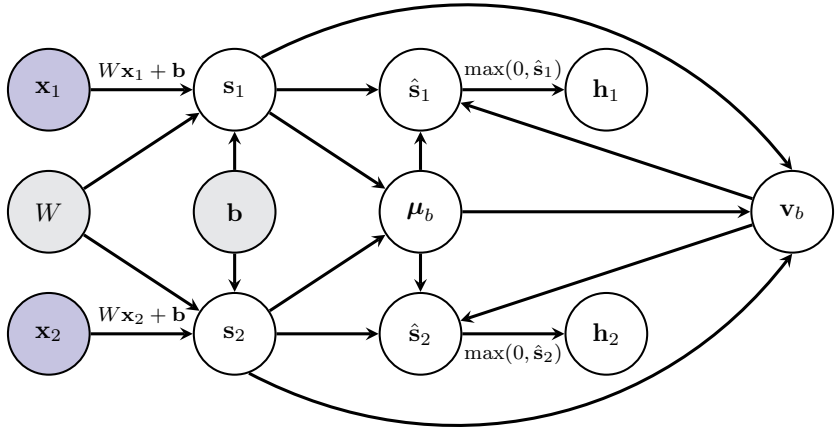
- Next $\frac{\partial \mathbf{v}_b}{\partial \mathbf{s}_i} = \frac{2}{n}\,\mathrm{diag}\,(\mathbf{s}_i - \boldsymbol{\mu}_b)$.
- As

$$v_{b,j} = \frac{1}{n}\sum_{l=1}^{n}\left(s_{l,j} - \mu_{b,j}\right)^2$$

and

$$\frac{\partial v_{b,j}}{\partial s_{i,k}} = \begin{cases} \frac{2}{n}\left(s_{i,j} - \mu_{b,j}\right) & \text{if } j = k \\ 0 & \text{otherwise} \end{cases}$$

Gradient Computations for a BN layer



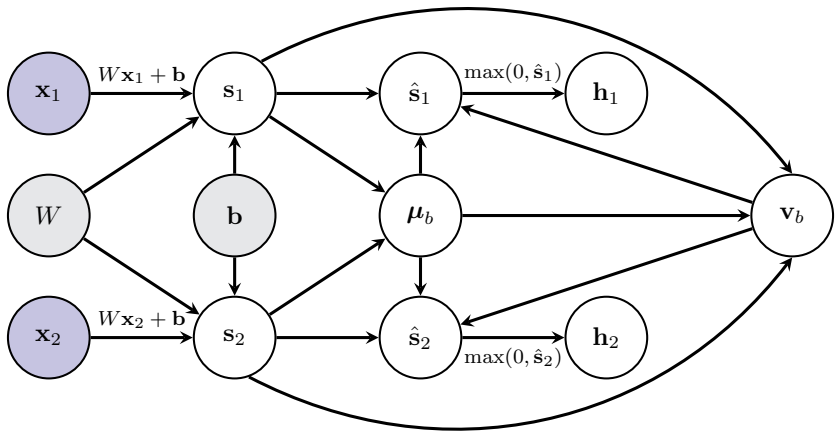
$$\frac{\partial J}{\partial \mathbf{s}_i} = \frac{\partial J}{\partial \hat{\mathbf{s}}_i}\frac{\partial \hat{\mathbf{s}}_i}{\partial \mathbf{s}_i} + \frac{\partial J}{\partial \mathbf{v}_b}\frac{\partial \mathbf{v}_b}{\partial \mathbf{s}_i} + \frac{\partial J}{\partial \boldsymbol{\mu}_b}\frac{\partial \boldsymbol{\mu}_b}{\partial \mathbf{s}_i}$$

- The children of node $\boldsymbol{\mu}_b$ are $\{\hat{\mathbf{s}}_1, \ldots, \hat{\mathbf{s}}_n, \mathbf{v}_b\}$.
- Therefore

$$\frac{\partial J}{\partial \boldsymbol{\mu}_b} = \sum_{i=1}^{n} \frac{\partial J}{\partial \hat{\mathbf{s}}_i}\frac{\partial \hat{\mathbf{s}}_i}{\partial \boldsymbol{\mu}_b} + \frac{\partial J}{\partial \mathbf{v}_b}\frac{\partial \mathbf{v}_b}{\partial \boldsymbol{\mu}_b}$$

Gradient Computations for a BN layer

$$\frac{\partial J}{\partial \boldsymbol{\mu}_b} = \sum_{i=1}^{n} \frac{\partial J}{\partial \hat{\mathbf{s}}_i} \frac{\partial \hat{\mathbf{s}}_i}{\partial \boldsymbol{\mu}_b} + \frac{\partial J}{\partial \mathbf{v}_b} \frac{\partial \mathbf{v}_b}{\partial \boldsymbol{\mu}_b}$$



- The equation relating $\hat{\mathbf{s}}_i$ to $\boldsymbol{\mu}_b$ (remember $V_b = \text{diag}(\mathbf{v}_b + \epsilon)$)

$$\hat{\mathbf{s}}_i = V_b^{-\frac{1}{2}} (\mathbf{s}_i - \boldsymbol{\mu}_b)$$

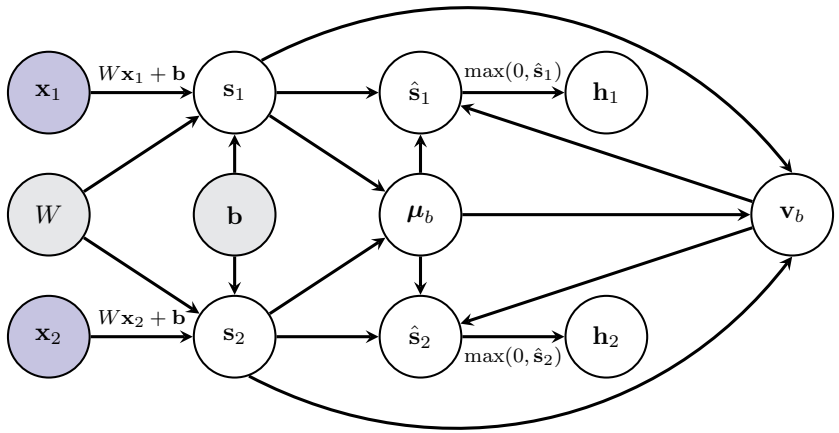
- The local Jacobian we want to compute

$$\frac{\partial \hat{\mathbf{s}}_i}{\partial \boldsymbol{\mu}_b} = -V_b^{-\frac{1}{2}}$$

Gradient Computations for a BN layer

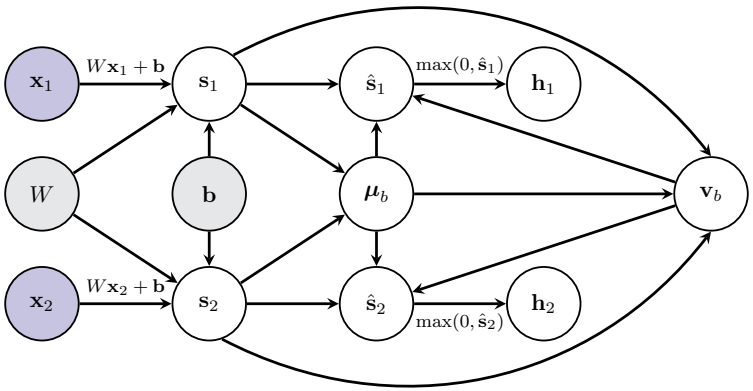
$$\frac{\partial J}{\partial \boldsymbol{\mu}_b} = \sum_{i=1}^{n} \frac{\partial J}{\partial \hat{\mathbf{s}}_i} \frac{\partial \hat{\mathbf{s}}_i}{\partial \boldsymbol{\mu}_b} + \frac{\partial J}{\partial \mathbf{v}_b} \frac{\partial \mathbf{v}_b}{\partial \boldsymbol{\mu}_b}$$

↑ already calculated



Gradient Computations for a BN layer

$$\frac{\partial J}{\partial \boldsymbol{\mu}_b} = \sum_{i=1}^{n} \frac{\partial J}{\partial \hat{\mathbf{s}}_i} \frac{\partial \hat{\mathbf{s}}_i}{\partial \boldsymbol{\mu}_b} + \frac{\partial J}{\partial \mathbf{v}_b} \frac{\partial \mathbf{v}_b}{\partial \boldsymbol{\mu}_b}$$



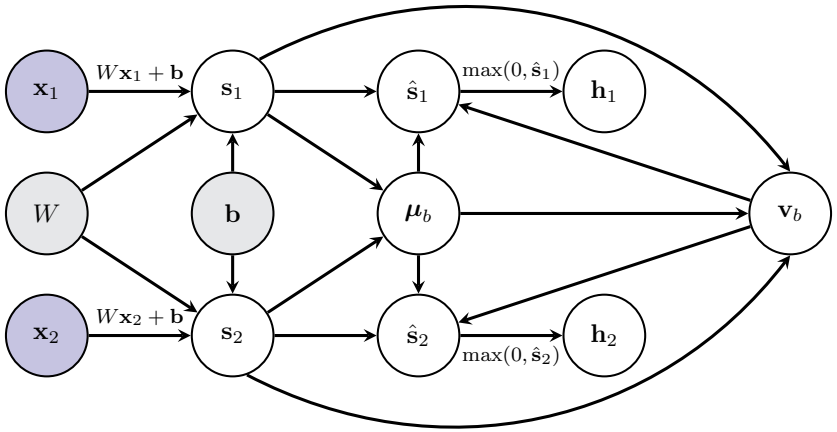
- Next $\frac{\partial \mathbf{v}_b}{\partial \boldsymbol{\mu}_b} = \mathbf{0}$.
- As

$$v_{b,j} = \frac{1}{n} \sum_{i=1}^{n} \left(s_{i,j} - \mu_{b,j}\right)^2$$

and

$$\frac{\partial v_{b,j}}{\partial \mu_{b,k}} = \begin{cases} -\frac{2}{n} \sum_{i=1}^{n} \left(s_{i,j} - \mu_{b,j}\right) = 0 & \text{if } j = k \\ 0 & \text{otherwise} \end{cases}$$

Gradient Computations for a BN layer



$$\frac{\partial J}{\partial \mathbf{s}_i} = \frac{\partial J}{\partial \hat{\mathbf{s}}_i}\frac{\partial \hat{\mathbf{s}}_i}{\partial \mathbf{s}_i} + \frac{\partial J}{\partial \mathbf{v}_b}\frac{\partial \mathbf{v}_b}{\partial \mathbf{s}_i} + \frac{\partial J}{\partial \boldsymbol{\mu}_b}\frac{\partial \boldsymbol{\mu}_b}{\partial \mathbf{s}_i}$$

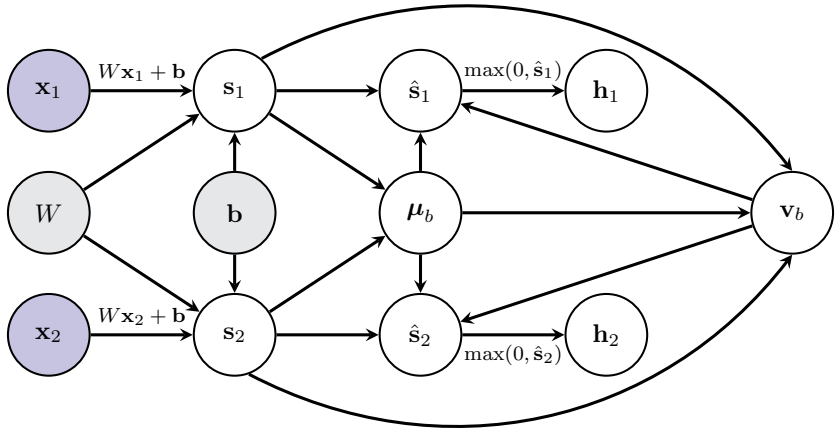
- The equation relating $\boldsymbol{\mu}_b$ to $\mathbf{s}_l$'s is

$$\boldsymbol{\mu}_b = \frac{1}{n}\sum_{l=1}^{n} \mathbf{s}_l$$

- Therefore

$$\frac{\partial \boldsymbol{\mu}_b}{\partial \mathbf{s}_i} = \frac{1}{n} I_m$$

Putting everything together



$$\frac{\partial J}{\partial \mathbf{v}_b} = -\frac{1}{2} \sum_{i=1}^{n} \frac{\partial J}{\partial \hat{\mathbf{s}}_i} V_b^{-\frac{3}{2}} \mathsf{diag}(\mathbf{s}_i - \boldsymbol{\mu}_b)$$

$$\frac{\partial J}{\partial \boldsymbol{\mu}_b} = -\sum_{i=1}^{n} \frac{\partial J}{\partial \hat{\mathbf{s}}_i} V_b^{-\frac{1}{2}}$$

$$\frac{\partial J}{\partial \mathbf{s}_i} = \frac{\partial J}{\partial \hat{\mathbf{s}}_i} V_b^{-\frac{1}{2}} + \frac{2}{n} \frac{\partial J}{\partial \mathbf{v}_b} \mathsf{diag}\left(\mathbf{s}_i - \boldsymbol{\mu}_b\right) + \frac{\partial J}{\partial \boldsymbol{\mu}_b} \frac{1}{n}$$