

# Lecture 4 - k-layer Neural Networks

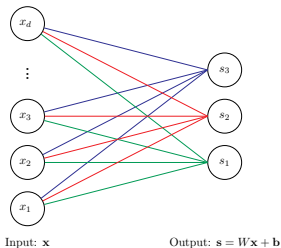
DD2424

April 12, 2017

# A new class of scoring functions

## Linear scoring function

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



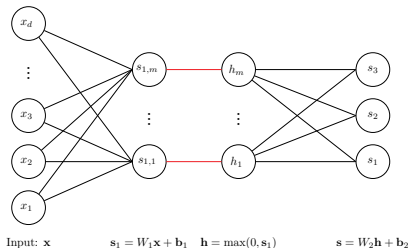
Before

## 2-layer Neural Network

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

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

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



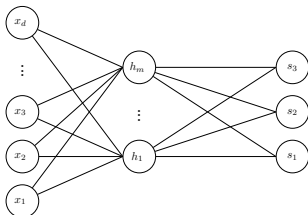
Now

## 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:  $\mathbf{x}$ 

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

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

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

## 3-layer Neural Network

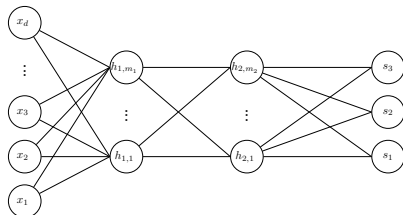
$$\mathbf{s}_1 = W_1 \mathbf{x} + \mathbf{b}_1$$

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

$$\mathbf{s}_2 = W_2 \mathbf{h}_1 + \mathbf{b}_2$$

$$\mathbf{h}_2 = \max(0, \mathbf{s}_2)$$

$$\mathbf{s} = W_3 \mathbf{h}_2 + \mathbf{b}_3$$

Input:  $\mathbf{x}$ 

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

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

$$\mathbf{s}_2 = W_2 \mathbf{h}_1 + \mathbf{b}_2$$

$$\mathbf{h}_2 = \max(0, \mathbf{s}_2)$$

Output:  $\mathbf{s} = W_3 \mathbf{h}_2 + \mathbf{b}_3$

## 3-layer Neural Network

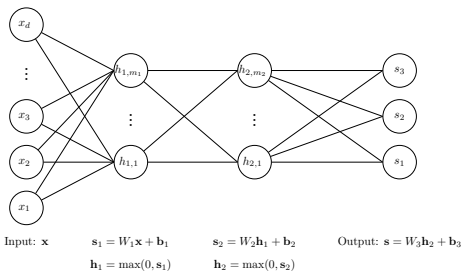
$$\mathbf{s}_1 = W_1 \mathbf{x} + \mathbf{b}_1 \quad W_1 \text{ is } m_1 \times d$$

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

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

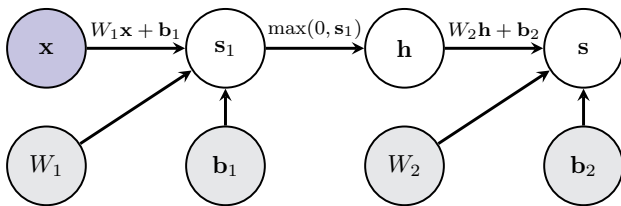
2nd hidden layer activations  $\rightarrow \mathbf{h}_2 = \max(0, \mathbf{s}_2) \leftarrow$  apply non-linearity via activation fn

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

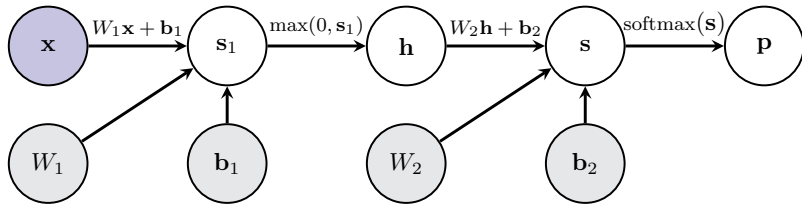


Sometimes referred to as a **2-hidden-layer neural network**.

# Computational Graph of our 2-layer neural network

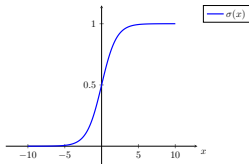


## 2-layer neural network with probabilistic outputs



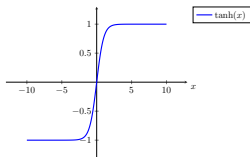
# Options for activation functions

## Sigmoid



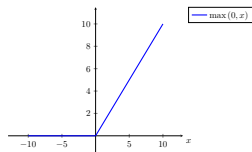
$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

## tanh



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

## ReLU

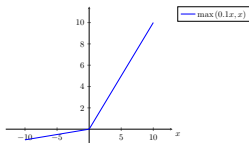


$$\text{ReLU}(x) = \max(0, x)$$

Activation function is applied independently to each element of the score vector.

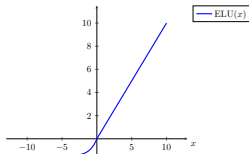
# Options for activation Functions

## Leaky ReLU



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

## ELU



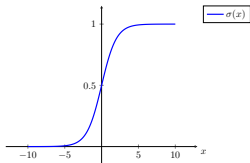
$$\text{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha (\exp(x) - 1) & \text{otherwise} \end{cases}$$

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



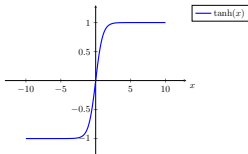
# Options for Activation Functions

## Sigmoid



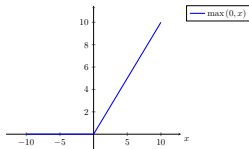
$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

## tanh



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

## ReLU

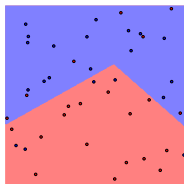


$$\text{ReLU}(x) = \max(0, x)$$

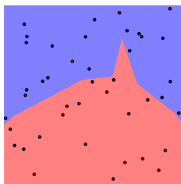


In modern networks ReLU is the most common activation function.

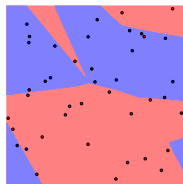
# Effect of the number of hidden nodes in a 2 layer network



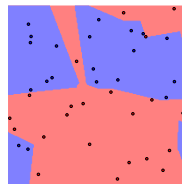
$m = 3$



$m = 20$



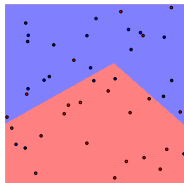
$m = 30$



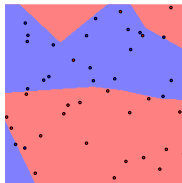
$m = 100$

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

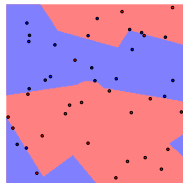
# Result depends on parameter initialization



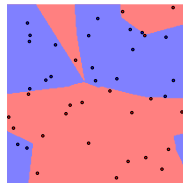
$m = 3$



$m = 20$



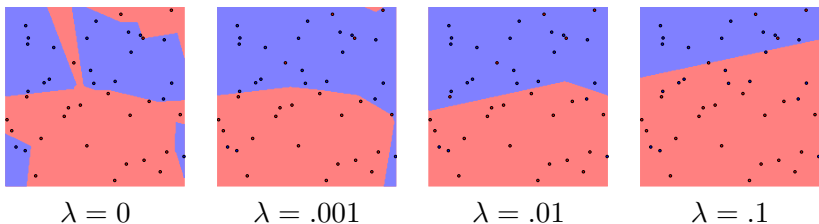
$m = 30$



$m = 100$

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

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



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

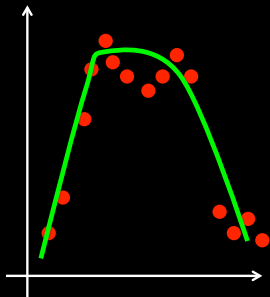
**Do not use size of neural network as a regularizer.**

**Use stronger regularization.**

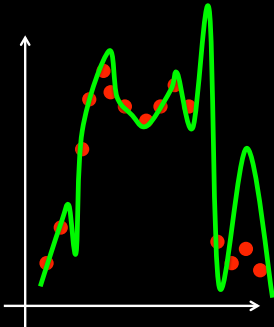
# Big Model + Regularize vs Small Model

---

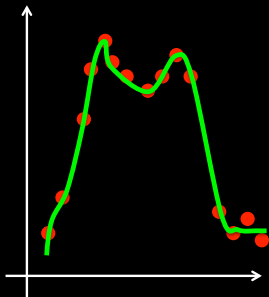
Small model



Big model



Big model  
+ Regularize



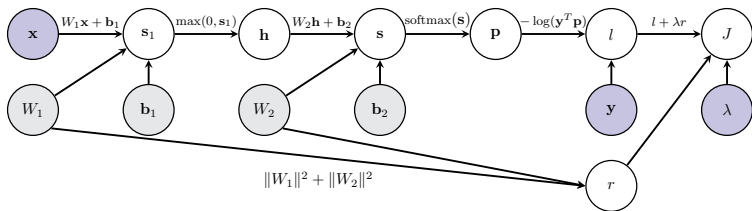
## Mini-batch SGD (or variant)

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

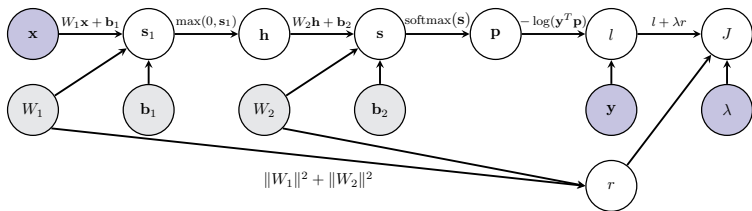


For a single labelled training example:

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:

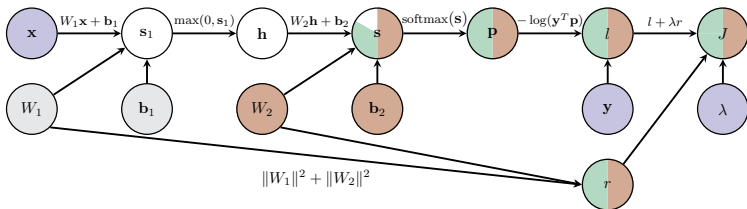
1. **Forward propagate** it through the graph and calculate loss.

↑ this is straightforward.

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

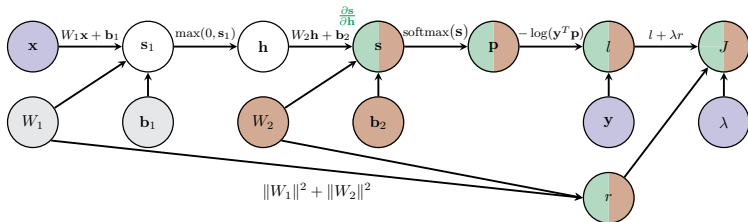
# Backward Pass: Gradient of current node

## Starting point of our demonstration



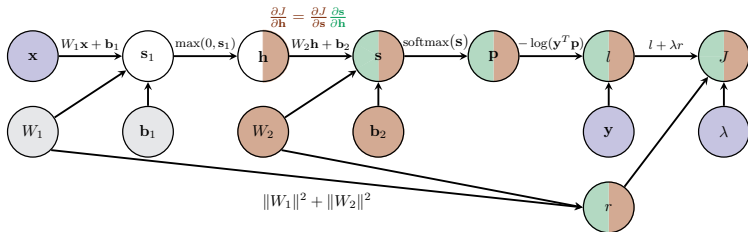
In Lecture 3 explicitly computed **filled in local Jacobians** and *gradients*.

## Compute local Jacobian of node $s$ w.r.t. its child $h$



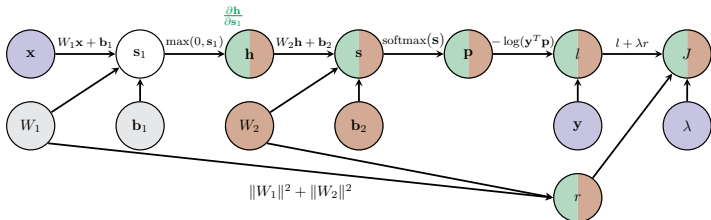
$$s = W_2 h + b_2$$

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

Compute gradient of  $J$  w.r.t. node  $h$ 

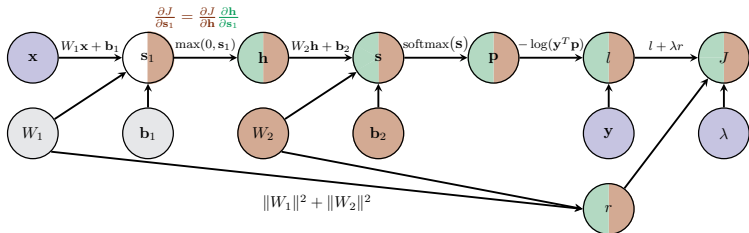
$$s = W_2h + b_2$$

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

Compute local Jacobian of node  $h$  w.r.t. its child  $s_1$ 

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

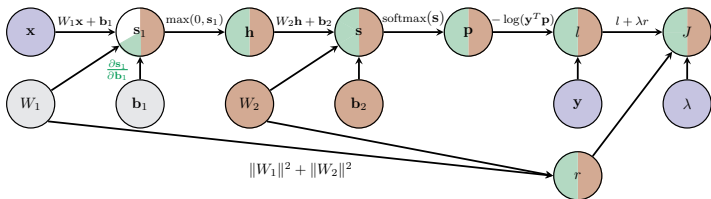
- The Jacobian we need to compute:  $\frac{\partial \mathbf{h}}{\partial \mathbf{s}_1} = \begin{pmatrix} \frac{\partial h_1}{\partial s_{1,1}} & \dots & \frac{\partial h_1}{\partial s_{1,m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_m}{\partial s_{1,1}} & \dots & \frac{\partial h_c}{\partial s_{1,m}}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

Compute gradient of  $J$  w.r.t. node  $s_1$ 

$$\mathbf{h} = \max(0, s_1)$$

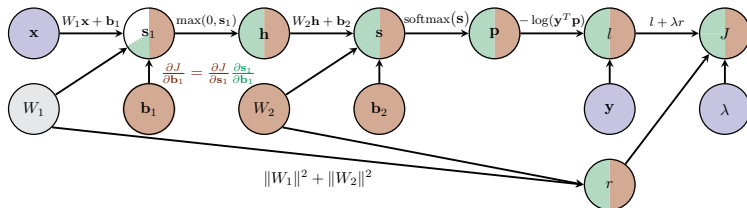
$$\frac{\partial J}{\partial s_1} = \frac{\partial J}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial s_1}$$

## Compute local Jacobian of node $s_1$ w.r.t. its child $b_1$



$$s_1 = W_1 x + b_1$$

- The Jacobian we need to compute:  $\frac{\partial s_1}{\partial b_1} = \begin{pmatrix} \frac{\partial s_{1,1}}{\partial b_{1,1}} & \cdots & \frac{\partial s_{1,1}}{\partial b_{1,m}} \\ \vdots & \vdots & \vdots \\ \frac{\partial s_{1,m}}{\partial b_{1,1}} & \cdots & \frac{\partial s_{1,m}}{\partial b_{1,m}} \end{pmatrix}$
- 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:  $\frac{\partial s_1}{\partial b_1} = I_m$

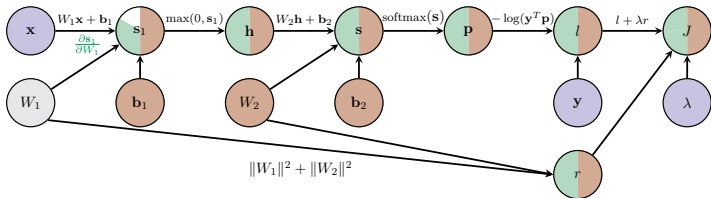
Compute gradient of  $J$  w.r.t. node  $b_1$ 

$$s_1 = W_1 x + b_1$$

$$\frac{\partial J}{\partial b_1} = \frac{\partial J}{\partial s_1} \frac{\partial s_1}{\partial b_1}$$



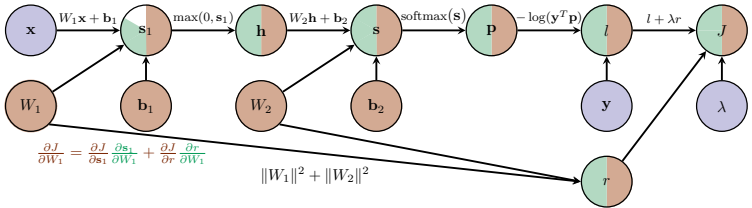
## Compute local Jacobian of node $s_1$ w.r.t. its child $W$



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

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

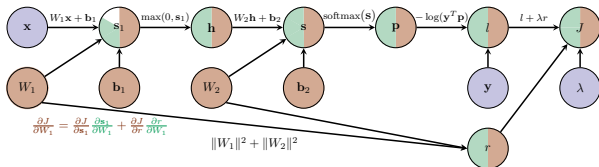
## Compute gradient of $J$ w.r.t. node $W_1$



$$s_1 = W_1 \mathbf{x} + \mathbf{b}_1 = (I_m \otimes \mathbf{x}^T) \text{vec}(W_1) + \mathbf{b}_1$$

$$\begin{aligned} \frac{\partial J}{\partial \text{vec}(W_1)} &= \frac{\partial J}{\partial s_1} \frac{\partial s_1}{\partial \text{vec}(W_1)} + \frac{\partial J}{\partial r} \frac{\partial r}{\partial \text{vec}(W_1)} \\ &= (g_1 \mathbf{x}^T \quad g_2 \mathbf{x}^T \quad \dots \quad g_m \mathbf{x}^T) + \lambda \text{vec}(W_1)^T \quad \leftarrow \text{gradient needed for learning} \end{aligned}$$

if we set  $\mathbf{g} = \frac{\partial J}{\partial s_1}$ .

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

$$s_1 = W_1 x + b_1 = (I_m \otimes x^T) \text{vec}(W_1) + b_1$$

Can convert

$$\frac{\partial J}{\partial \text{vec}(W_1)} = (g_1 x^T \quad g_2 x^T \quad \dots \quad g_m x^T) + 2\lambda \text{vec}(W_1)^T$$

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

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

# Aggregated backward pass for a 2-layer neural network

1. Let

$$\mathbf{g} = -\frac{\mathbf{y}^T}{\mathbf{y}^T \mathbf{p}} \left( \text{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^T \right)$$

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. Propagate the gradient vector  $\mathbf{g}$  to the first layers

$$\mathbf{g} = \mathbf{g}W_2$$

$$\mathbf{g} = \mathbf{g} \text{diag}(\text{Ind}(\mathbf{s}_1 > 0)) \leftarrow \text{assuming ReLU activation}$$

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  $(\mathbf{x}, y) \in \mathcal{D}^{(t)}$

1. Let  $\mathbf{g} = -\frac{\mathbf{y}^T}{\mathbf{y}^T \mathbf{p}} (\text{diag}(\mathbf{p}) - \mathbf{p} \mathbf{p}^T)$

2. Add gradient of  $l$  w.r.t.  $\mathbf{b}_2$  computed at  $(\mathbf{x}, y)$

$$\frac{\partial L}{\partial \mathbf{b}_2} += \mathbf{g}, \quad \frac{\partial L}{\partial W_2} += \mathbf{g}^T \mathbf{h}^T$$

3. Propagate the gradients

$$\mathbf{g} = \mathbf{g} W_2$$

$$\mathbf{g} = \mathbf{g} \text{diag}(\text{ln}(\mathbf{s}_1 > 0))$$

4. Add gradient of  $l$  w.r.t. first layer parameters computed at  $(\mathbf{x}, y)$

$$\frac{\partial L}{\partial \mathbf{b}_1} += \mathbf{g}, \quad \frac{\partial L}{\partial W_1} += \mathbf{g}^T \mathbf{x}^T$$

- Divide by the number of entries in  $\mathcal{D}^{(t)}$ :

$$\frac{\partial L}{\partial W_i} /= |\mathcal{D}^{(t)}|, \quad \frac{\partial L}{\partial \mathbf{b}_i} /= |\mathcal{D}^{(t)}| \quad \text{for } i = 1, 2$$

- Add the gradient for the regularization term

$$\frac{\partial J}{\partial W_i} = \frac{\partial L}{\partial W_i} + 2\lambda W_i, \quad \frac{\partial J}{\partial \mathbf{b}_i} = \frac{\partial L}{\partial \mathbf{b}_i} \quad \text{for } i = 1, 2$$

# Forward pass for a k-layer neural network

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

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

$$\mathbf{x}^{(i)} = \max(0, \mathbf{s}^{(i)})$$

- 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  $\mathbf{y}$

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

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

$$\mathbf{g} = -\frac{\mathbf{y}^T}{\mathbf{y}^T \mathbf{p}} \left( \text{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^T \right)$$

- for  $i = k, k-1, \dots, 1$

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

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

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

Training Neural Networks a little 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  $20 \times 20$  pixel image and recognized letters.
- Perceptron classification fn:

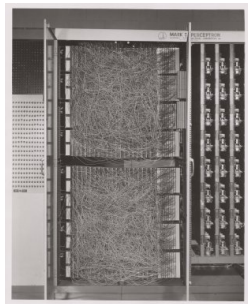
$$f(\mathbf{x}; \mathbf{w}) = \begin{cases} 1 & \text{if } \mathbf{w}^T \mathbf{x} + \mathbf{b} > 0 \\ 0 & \text{otherwise} \end{cases}$$

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

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

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

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



- 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  $(\mathbf{x}, y)$

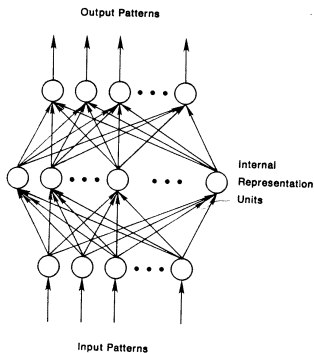
$$l(\mathbf{x}, y, \mathbf{w}) = (y - (\mathbf{w}^T \mathbf{x} + b))^2 = (y - \hat{y})^2$$

- **Update rule:** Use SGD with learning rate  $\eta$  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.

*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

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

be our measure of the error on input/output pattern  $p$  and let  $E = \sum_p E_p$  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

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

which is proportional to  $\Delta_j w_{ji}$  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.

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

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

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

Not surprisingly, the contribution of unit  $j$  to the error is simply proportional to  $\delta_{pj}$ . Moreover, since we have linear units,

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

from which we conclude that

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

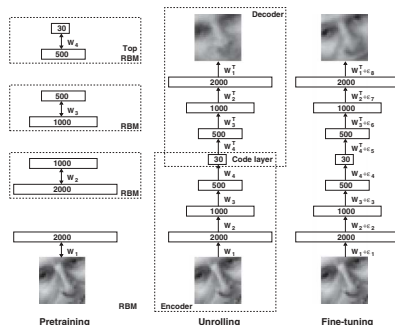
Thus, substituting back into Equation 3, we see that

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

**First time back-propagation became popular**

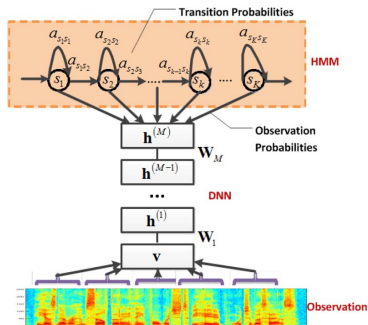
## New wave of research in Deep Learning.

- Ability to train networks with many layers.
- Mixture of unsupervised and supervised training.
- **Unsupervised:** Encoding layers first learnt in stagewise manner using RBMs (restricted Boltzman machines).
- Decode layers using an auto-encoder.
- **Supervised:** Back-prop used to do final update of weights.



# First Very Convincing Results

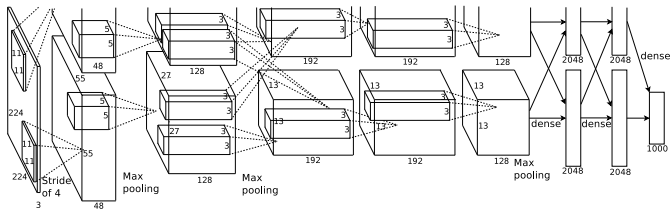
- **Context-Dependent Pre-trained Deep Neural Networks for Large Vocabulary Speech Recognition**, G. Dahl, D. Yu, L. Deng, A. Acero, 2010.



- Beat the widely established approach of GMM-HMM with a DNN-HMM.
- Improved results on popular datasets by 5.8% and 9.2%.

# First Very Convincing Results

- **ImageNet classification with deep convolutional neural networks A.** Krizhevsky, I. Sutskever, G. Hinton, 2012.

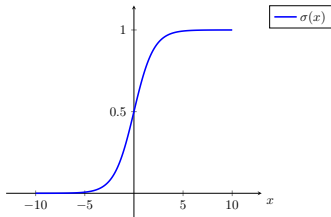


- Beat the stagnating established approaches of *Handcrafted features + kernel SVM*.
- Improved results on ImageNet by  $\sim 11\%$ .

Better understanding of gradient flows during BackProp helped with these breakthroughs

**Understanding Effect of Activation Functions**

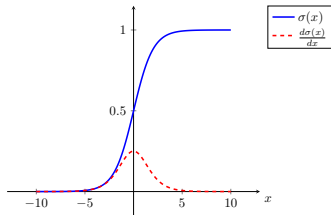
$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$



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



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



## 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:

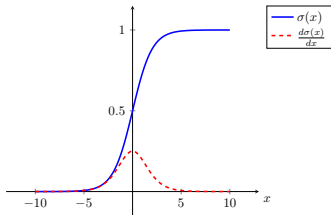
$$\frac{\partial h_i}{\partial s_j} = \begin{cases} \frac{\exp(-s_i)}{(1 + \exp(-s_i))^2} \quad (= \sigma'(s_i)) & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

- 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 gradient of  $J$  w.r.t.  $s_i$  when  $|s_i| > 5$ ?

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



## Problems

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

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

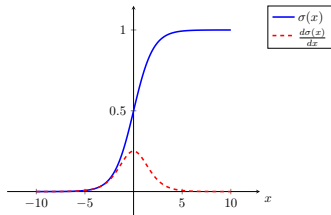
$$\mathbf{h} = \sigma(\mathbf{s})$$

- 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?

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



## Problems

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

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

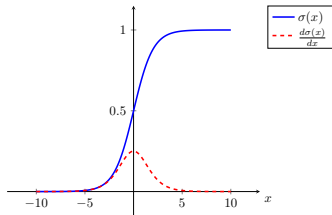
- Then

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

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

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

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



## Problems

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

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

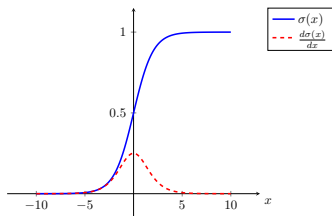
- Then

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

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

- $\implies$  entries of  $\frac{\partial J}{\partial \mathbf{w}_i}$  are either all positive or all negative.
- $\implies$  inefficient zig-zag update paths to find optimal  $\mathbf{w}_i$

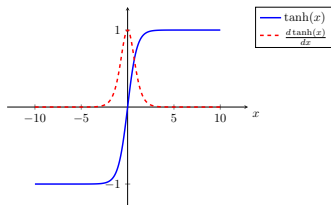
$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$



## Problems

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

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

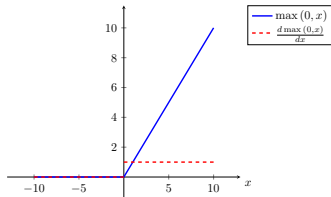


## Properties

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

# Rectified Linear Unit (ReLU)

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

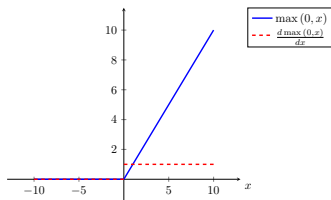


## Pros

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:

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

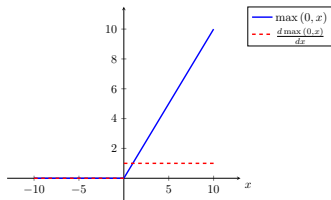
- Then

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



# Rectified Linear Unit (ReLU)

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



## Problems

1. Output is not zero-centered
2. Negative activations have zero gradients and freezes some parameter weights.

As

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

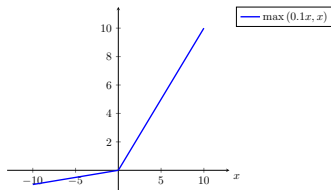
then

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

$\implies$  dead ReLU will never activate

$\implies$  never update parameter weights.

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



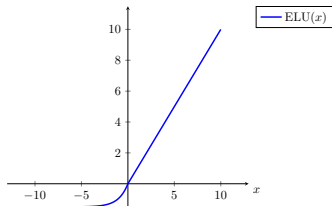
## Pros

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  $\exp()$

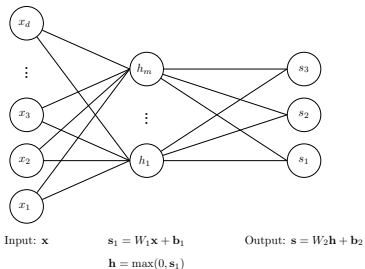
[Clevert et al., 2015]

## In practice

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

Effect of weight initialization & activation function on gradient flow

## 2-layer Neural Network



What happens when you initialize each weight matrix entry to zero? (each  $W_{i,lm} = 0$ )

# Initialize with small random numbers

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

What happens in this case?

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

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.

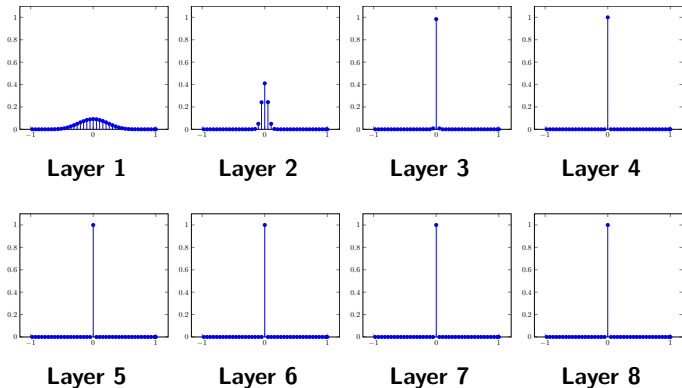


## Some activation histograms

- Initialize a 10-layer network with 500 nodes at each layer.
- Use a  $\tanh$  activation function at each layer.
- Initialize weights with small random numbers.
- Generate random input data ( $N(0, 1^2)$ ) with  $d = 500$ .

# Some activation histograms

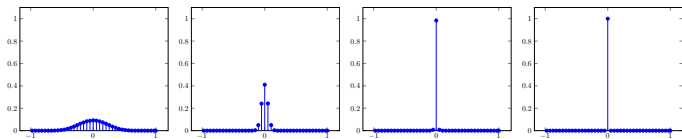
- Initialize a 10-layer network with 500 nodes at each layer.
- Use a  $\tanh$  activation function at each layer.
- Initialize weights with small random numbers.
- Generate random input data ( $N(0, 1^2)$ ) with  $d = 500$ .



Histograms of activations at each layer

# Some activation histograms

- All activations become zero at the layers  $> 2$ .
- What happens in the backward pass of the back-prop algorithm?

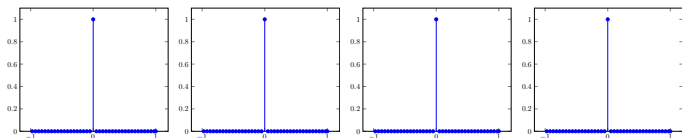


Layer 1

Layer 2

Layer 3

Layer 4



Layer 5

Layer 6

Layer 7

Layer 8

Histograms of activations at each layer

# Aggregated Backward pass for a k-layer neural network

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

- Let

$$\mathbf{g} = -\frac{\mathbf{y}^T}{\mathbf{y}^T \mathbf{p}} \left( \text{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^T \right)$$

- for  $i = k, k-1, \dots, 1$

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  $W_i$

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

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

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

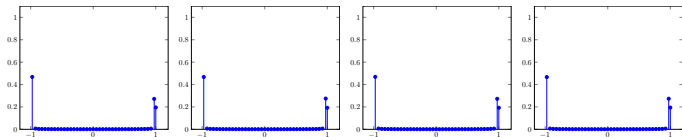
$$\mathbf{g} = \mathbf{g} \text{diag}(\text{ln}d(\mathbf{s}^{(i)} > 0))$$

# Change the initialization to bigger random numbers

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

# Change the initialization to bigger random numbers

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

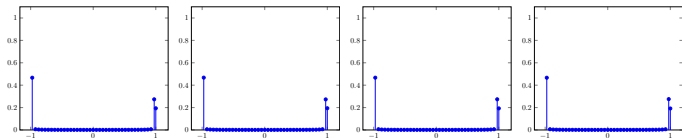


**Layer 1**

**Layer 2**

**Layer 3**

**Layer 4**



**Layer 5**

**Layer 6**

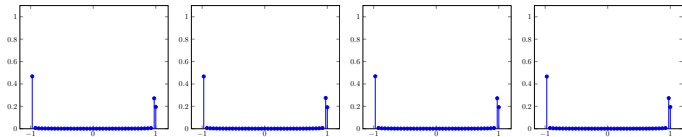
**Layer 7**

**Layer 8**

Histograms of activations at each layer

# Change the initialization to bigger random numbers

- Almost all neurons completely saturated, either  $-1$  or  $+1$ .
- $\implies$  Gradients will be all zero
- (Remember the picture of the gradient of  $\tanh$ .)

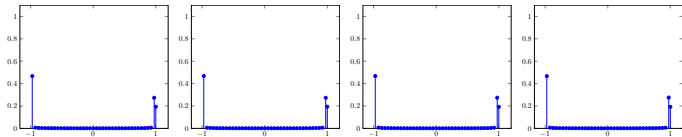


Layer 1

Layer 2

Layer 3

Layer 4



Layer 5

Layer 6

Layer 7

Layer 8

Histograms of activations at each layer

# Aggregated Backward pass for a k-layer neural network

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

- Let

$$\mathbf{g} = -\frac{\mathbf{y}^T}{\mathbf{y}^T \mathbf{p}} \left( \text{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^T \right)$$

- for  $i = k, k-1, \dots, 1$

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  $W_i$

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

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

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

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

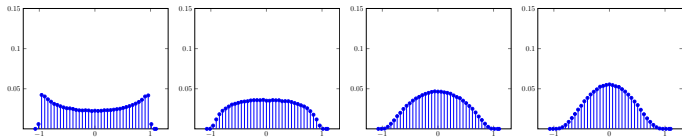


# Change the initialization to Xavier initialization

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

# Change the initialization to Xavier initialization

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

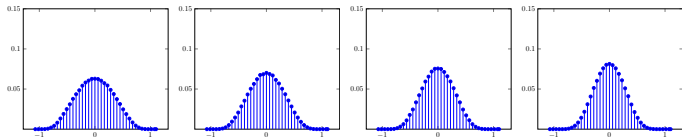


**Layer 1**

**Layer 2**

**Layer 3**

**Layer 4**



**Layer 5**

**Layer 6**

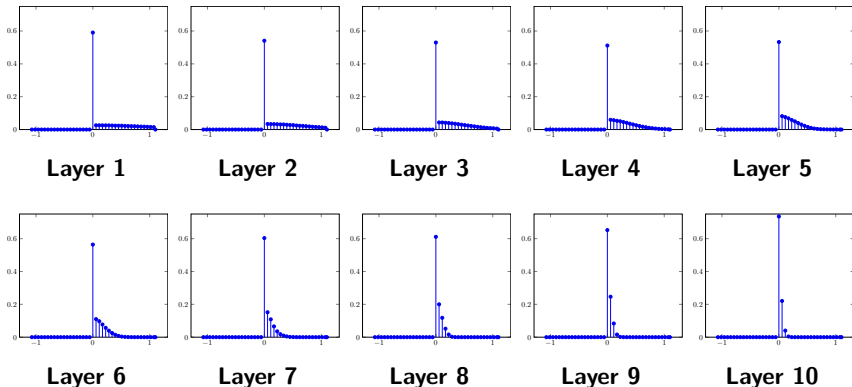
**Layer 7**

**Layer 8**

Histograms of activations at each layer

# Xavier initialization doesn't work for ReLu activation

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



Histograms of activations 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

- Want unit Gaussian activations at each layer?  
Just make them unit Gaussian!

- 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:  $\mathbf{s}_1^{(j)}, \mathbf{s}_2^{(j)}, \dots, \mathbf{s}_n^{(j)}$
- To make each dimension unit gaussian, apply:

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

where

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

- Usually apply **normalization** after the fully connected layer before non-linearity.
- Therefore for a  $k$ -layer network have
  - for  $i = 1, \dots, k - 1$   
for  $(\mathbf{x}^{(i-1)}, y) \in \mathcal{D}$  ← Apply  $i$ th linear transformation to batch

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

end

Compute batch mean and variances of  $i$ th layer:

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

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

$$\hat{\mathbf{s}}^{(i)} = \text{BatchNormalise}(\mathbf{s}^{(i)}, \boldsymbol{\mu}, \sigma_1, \dots, \sigma_{m_i})$$

$$\mathbf{x}^{(i)} = \max(0, \hat{\mathbf{s}}^{(i)})$$

end

end

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

# Batch Normalization: Scale & shift range

- Can also allow the network to squash and shift the range

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

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

- Can learn the  $\gamma^{(i)}$ 's and  $\beta^{(i)}$ 's and add them as parameters of the network.
- To keep things simple this added complexity is often omitted.



# Benefits of Batch Normalization

- Improves gradient flow through the network.
- Reduces the strong dependence on initialization.
- $\implies$  learn deeper networks more reliably.
- Allows higher learning rates.
- Acts as a form of regularization.

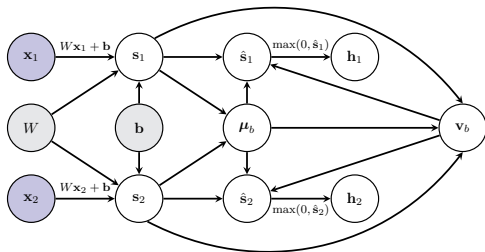
If training a deep network, you should use **Batch Normalization**.

# 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).

Back-Prop for a Batch Normalization layer.

# Computational Graph for a BN layer



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

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

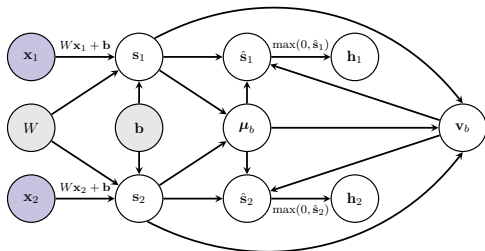
where  $\mathbf{v}_b = (v_{b,1}, v_{b,2}, \dots, v_{b,m})^T$ . ( $n = 2$  in the figure.) Define

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

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

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

# Gradient Computations for a BN layer

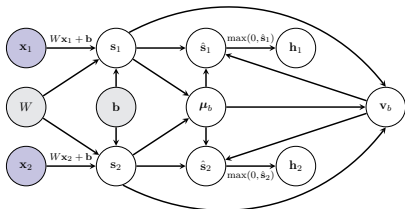


- Want to compute  $\frac{\partial J}{\partial s_i}$  for each  $s_i$  in the batch.
- The children of node  $s_i$  are  $\{\hat{s}_i, v_b, \mu_b\}$  thus

$$\frac{\partial J}{\partial s_i} = \frac{\partial J}{\partial \hat{s}_i} \frac{\partial \hat{s}_i}{\partial s_i} + \frac{\partial J}{\partial v_b} \frac{\partial v_b}{\partial s_i} + \frac{\partial J}{\partial \mu_b} \frac{\partial \mu_b}{\partial s_i}$$

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

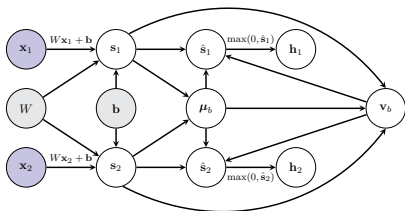
# Gradient Computations for a BN layer



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

↑  
assume already computed

# Gradient Computations for a BN layer



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

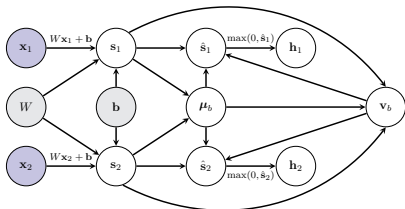
- The equation relating  $\hat{\mathbf{s}}_i$  to  $\mathbf{v}_b$  (remember  $V_b = \text{diag}(\mathbf{v}_b + \epsilon)$ )

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

- Therefore

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

# Gradient Computations for a BN layer



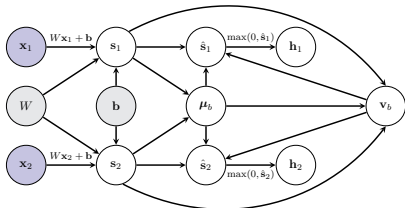
$$\frac{\partial J}{\partial s_i} = \frac{\partial J}{\partial \hat{s}_i} \frac{\partial \hat{s}_i}{\partial s_i} + \frac{\partial J}{\partial v_b} \frac{\partial v_b}{\partial s_i} + \frac{\partial J}{\partial \mu_b} \frac{\partial \mu_b}{\partial s_i}$$

- The children of node  $v_b$  are  $\{\hat{s}_1, \dots, \hat{s}_n\}$
- Therefore

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



# Gradient Computations for a BN layer



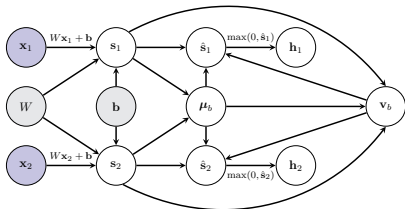
$$\frac{\partial J}{\partial \mathbf{s}_i} = \frac{\partial J}{\partial \hat{\mathbf{s}}_i} \frac{\partial \hat{\mathbf{s}}_i}{\partial \mathbf{s}_i} + \frac{\partial J}{\partial \mathbf{v}_b} \frac{\partial \mathbf{v}_b}{\partial \mathbf{s}_i} + \frac{\partial J}{\partial \mu_b} \frac{\partial \mu_b}{\partial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

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

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

↑  
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  $\{\hat{\mathbf{s}}_1, \dots, \hat{\mathbf{s}}_n\}$
- Therefore

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

↑  
compute now

# Gradient Computations for a BN layer

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

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

- The local Jacobian we want to compute

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

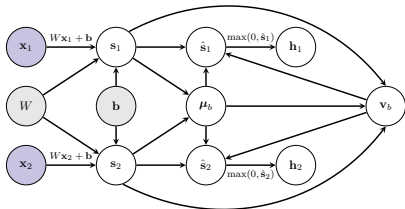
- Computing the derivative for each individual element:

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

- In matrix form

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

# Gradient Computations for a BN layer



$$\frac{\partial J}{\partial s_i} = \frac{\partial J}{\partial \hat{s}_i} \frac{\partial \hat{s}_i}{\partial s_i} + \frac{\partial J}{\partial v_b} \frac{\partial v_b}{\partial s_i} + \frac{\partial J}{\partial \mu_b} \frac{\partial \mu_b}{\partial s_i}$$

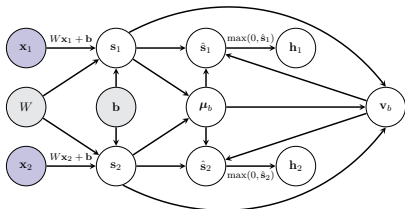
- Next  $\frac{\partial v_b}{\partial s_i} = \frac{2}{n} \text{diag}(s_i - \mu_b)$ .
- As

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

and

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

# Gradient Computations for a BN layer



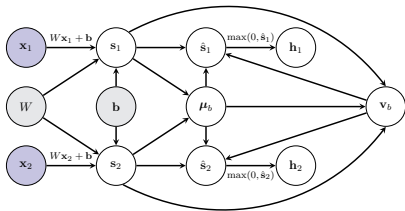
$$\frac{\partial J}{\partial s_i} = \frac{\partial J}{\partial \hat{s}_i} \frac{\partial \hat{s}_i}{\partial s_i} + \frac{\partial J}{\partial v_b} \frac{\partial v_b}{\partial s_i} + \frac{\partial J}{\partial \mu_b} \frac{\partial \mu_b}{\partial s_i}$$

- The children of node  $\mu_b$  are  $\{\hat{s}_1, \dots, \hat{s}_n, v_b\}$ .
- Therefore

$$\frac{\partial J}{\partial \mu_b} = \sum_{i=1}^n \frac{\partial J}{\partial \hat{s}_i} \frac{\partial \hat{s}_i}{\partial \mu_b} + \frac{\partial J}{\partial v_b} \frac{\partial v_b}{\partial \mu_b}$$

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

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

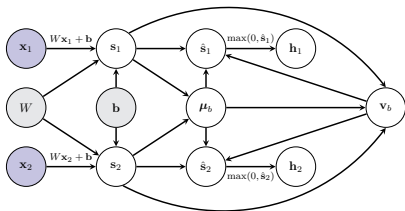
- The local Jacobian we want to compute

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

# Gradient Computations for a BN layer

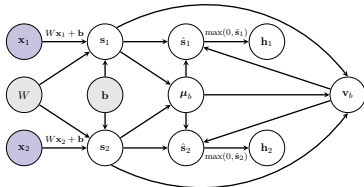
$$\frac{\partial J}{\partial \mu_b} = \sum_{i=1}^n \frac{\partial J}{\partial \hat{s}_i} \frac{\partial \hat{s}_i}{\partial \mu_b} + \frac{\partial J}{\partial \mathbf{v}_b} \frac{\partial \mathbf{v}_b}{\partial \mu_b}$$

↑  
already calculated



# Gradient Computations for a BN layer

$$\frac{\partial J}{\partial \mu_b} = \sum_{i=1}^n \frac{\partial J}{\partial \hat{s}_i} \frac{\partial \hat{s}_i}{\partial \mu_b} + \frac{\partial J}{\partial v_b} \frac{\partial v_b}{\partial \mu_b}$$



- Next  $\frac{\partial v_b}{\partial \mu_b} = 0$ .
- As

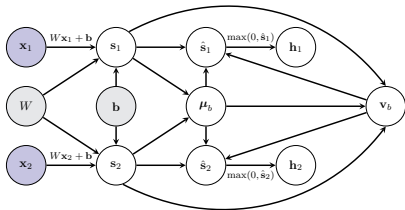
$$v_{b,j} = \frac{1}{n} \sum_{i=1}^n (s_{i,j} - \mu_{b,j})^2$$

and

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



# Gradient Computations for a BN layer



$$\frac{\partial J}{\partial s_i} = \frac{\partial J}{\partial \hat{s}_i} \frac{\partial \hat{s}_i}{\partial s_i} + \frac{\partial J}{\partial v_b} \frac{\partial v_b}{\partial s_i} + \frac{\partial J}{\partial \mu_b} \frac{\partial \mu_b}{\partial s_i}$$

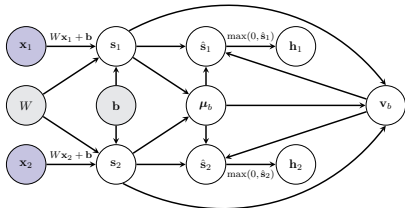
- The equation relating  $\mu_b$  to  $s_l$ 's is

$$\mu_b = \frac{1}{n} \sum_{l=1}^n s_l$$

- Therefore

$$\frac{\partial \mu_b}{\partial s_i} = \frac{1}{n} I_m$$

# Putting everything together



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

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

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