HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Lesson 3: Training neural networks

(Part 1)

Viet-Trung Tran

Outline

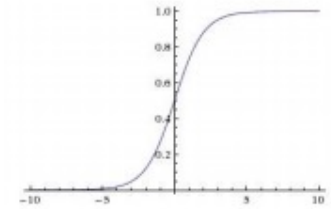
- Activation functions
- Data preprocessing
- Weight initializations
- Normalizations

Activation functions

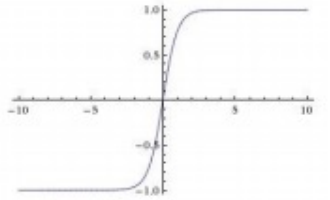
Activation Functions

Sigmoid

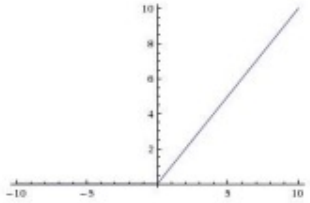
$$\sigma(x) = 1/(1+e^{-x})$$



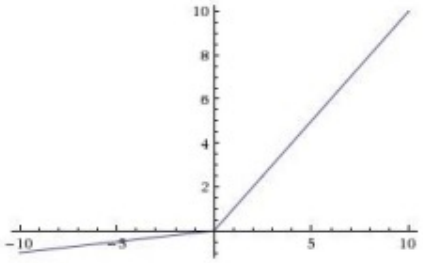
tanh tanh(x)



ReLU max(0,x)



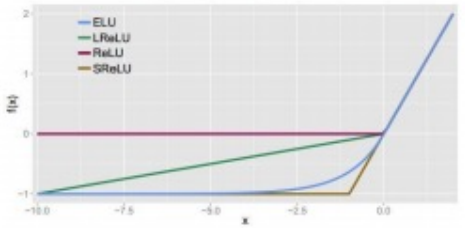
Leaky ReLU
max(0.1x, x)



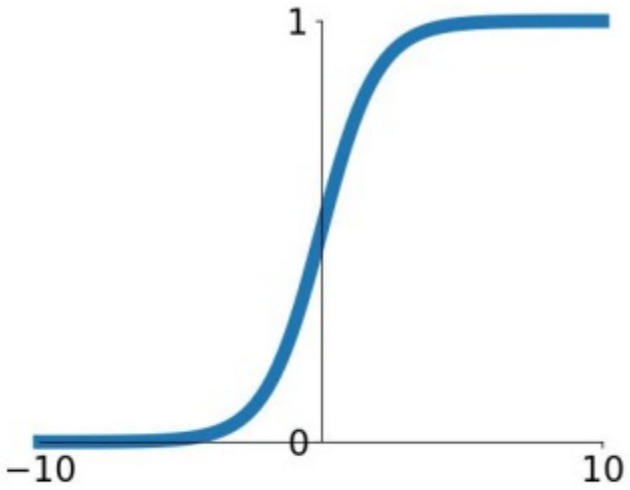
Maxout $\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

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



Sigmoid function

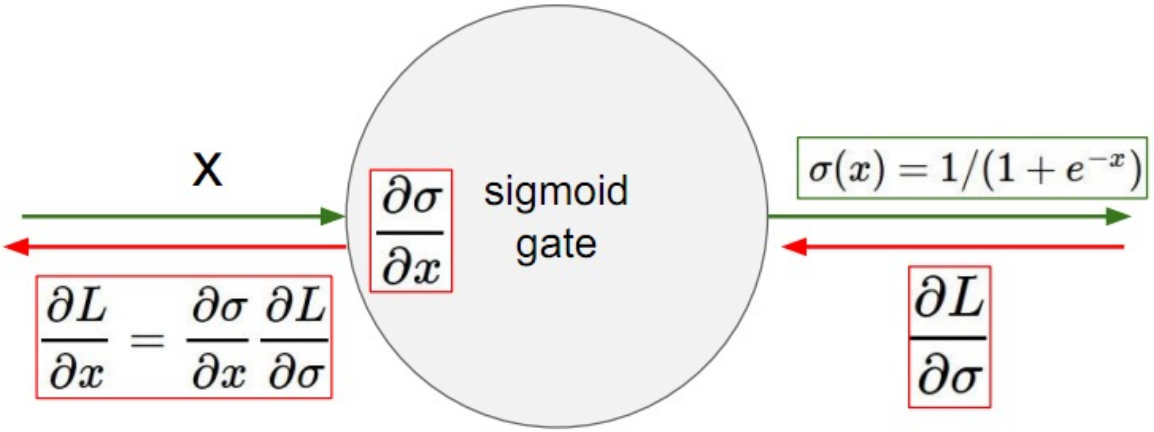


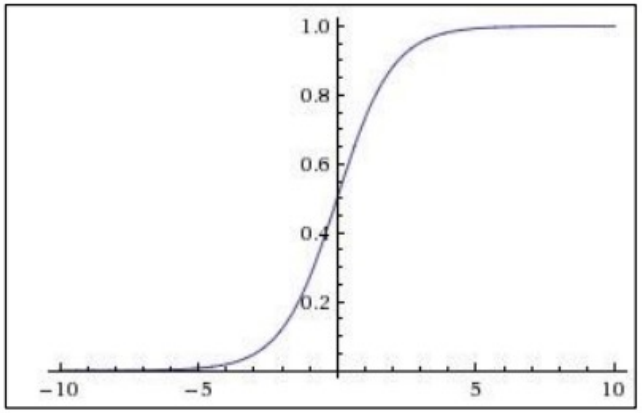
Sigmoid

$$\sigma(x) = 1/(1 + e^{-x})$$

- Squashes numbers to range [0,1]
- They are widely used in the history of neural networks because they simulate the firing rate of neurons.
- There are 3 downsides:
  1. Vanishing gradient due to saturated neurons

Sigmoid function (2)

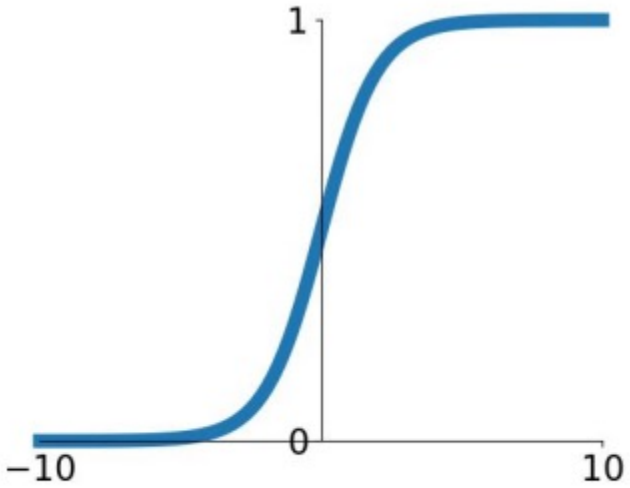




$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

- What happens when x = -10?
- What happens when x = 0?
- What happens when x = 10?

Sigmoid function (3)



Sigmoid

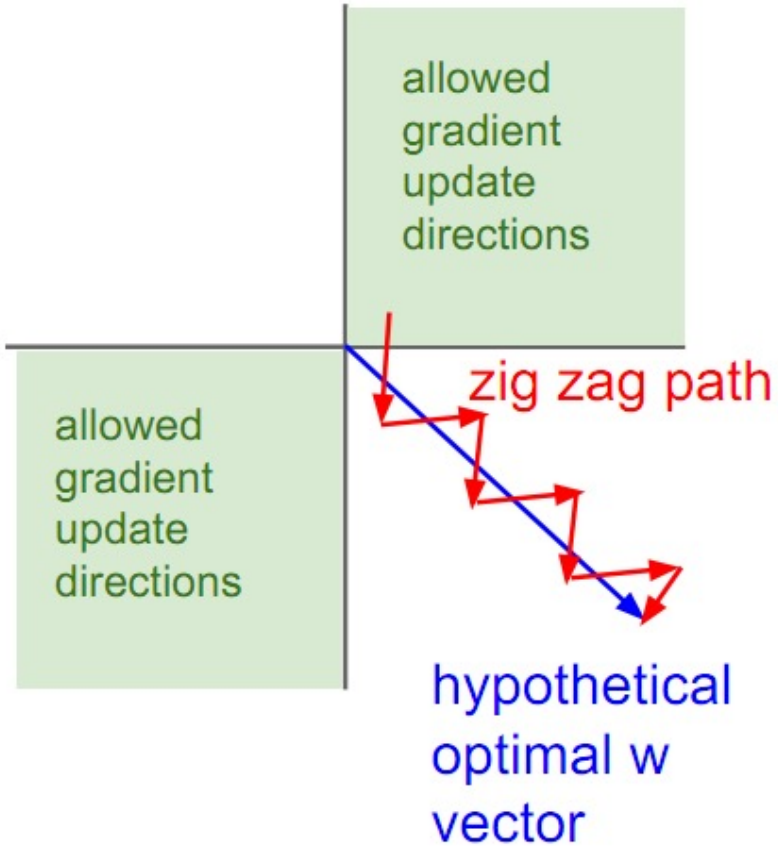
$$\sigma(x) = 1/(1 + e^{-x})$$

- Squashes numbers to range [0,1]
- They are widely used in the history of neural networks because they simulate the firing rate of neurons.
- There are 3 downsides:
  1. Vanishing gradient due to saturated neurons
  2. Output average is non-zero

Sigmoid function (4)

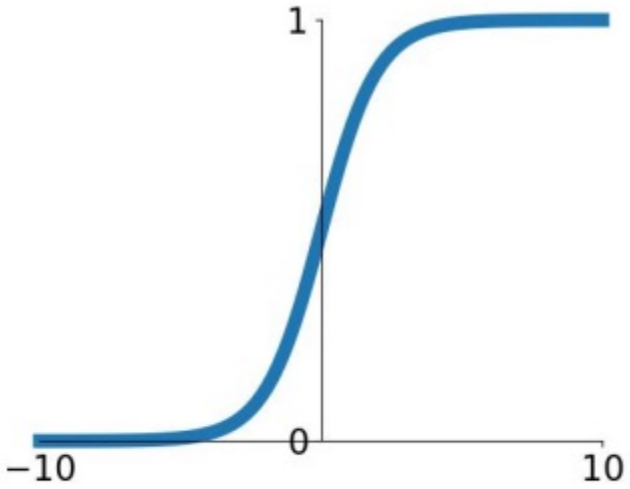
$$f\left(\sum_i w_i x_i + b\right)$$

- What if all the neuron's inputs $\mathbf{x_i}$ are positive?
- Then what will be the gradient of the objective function with respect to **w**?
- All the elements of **w** have the same sign as **f'(w)**, i.e. the same negative or the same positive.
- Then the gradient can only be directed in certain dimensions in the search space.



$$\frac{\partial L}{\partial w} = \sigma\left(\sum_i w_i x_i + b\right)\left(1 - \sigma\left(\sum_i w_i x_i + b\right)\right) x \times upstream\_gradient$$

Sigmoid function (5)



Sigmoid

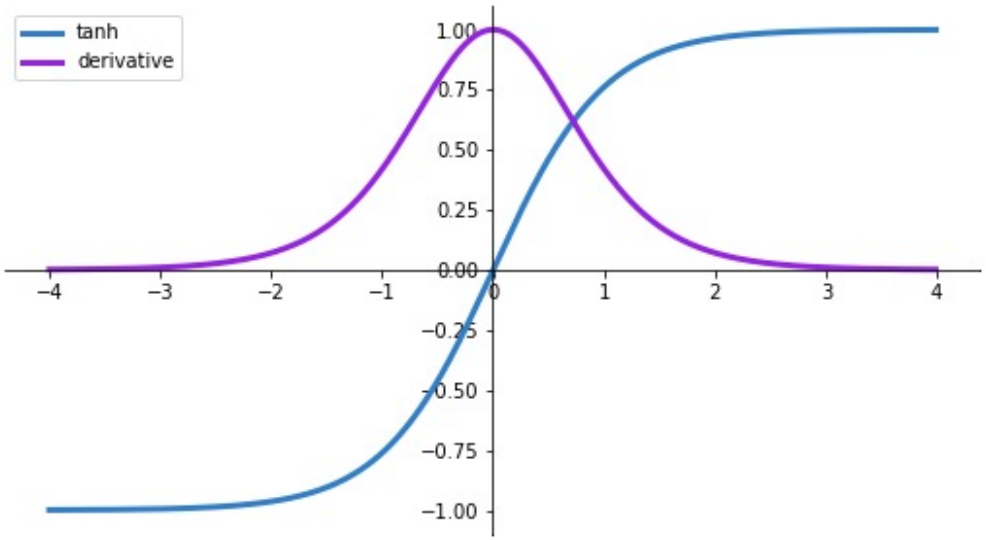
$$\sigma(x) = 1/(1 + e^{-x})$$

- Squashes numbers to range [0,1]
- They are widely used in the history of neural networks because they simulate the firing rate of neurons.
- There are 3 downsides:
  1. Vanishing gradient due to saturated neurons
  2. Output average is non-zero
  3. Calculating the exponential exp() is expensive

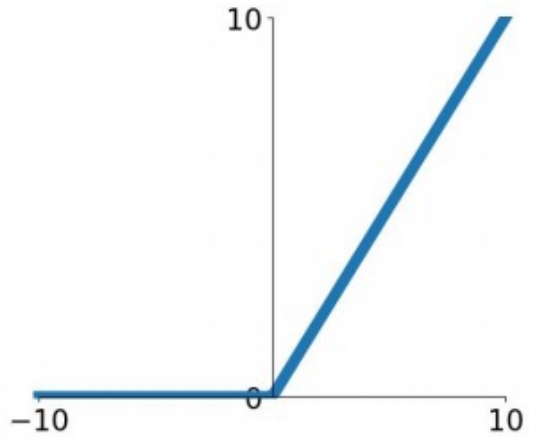
Tanh function

$$\frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- Squashes numbers to range [-1,1]
- Average output is 0
- Still suffering from saturated neurons, vanishing gradient



ReLU function

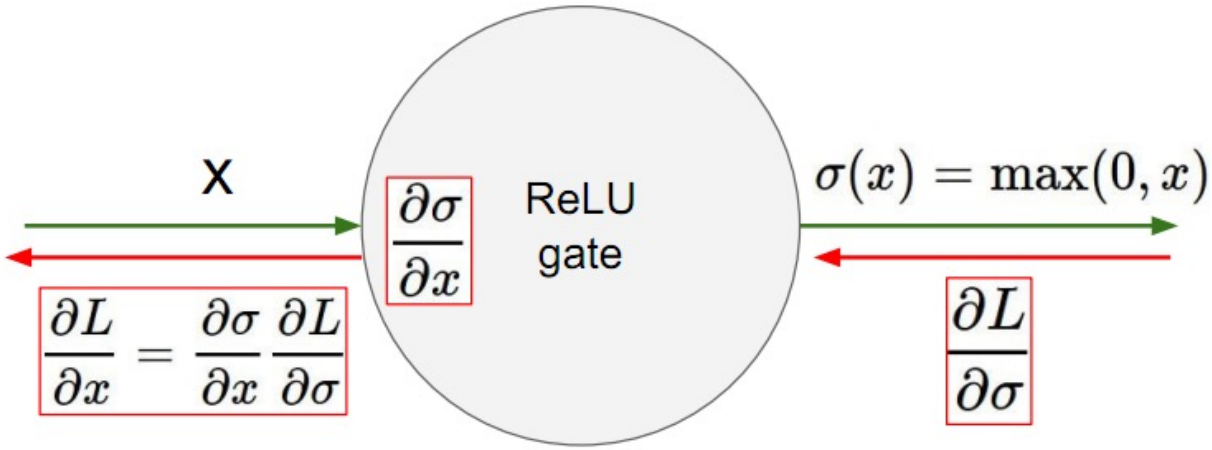


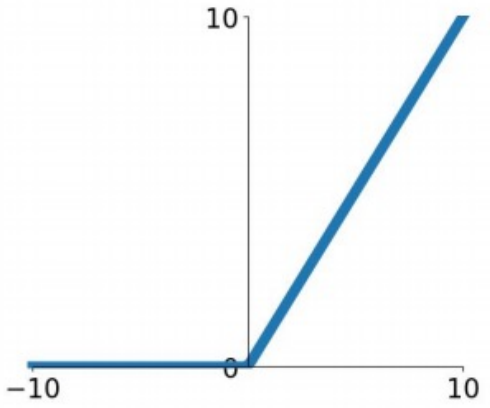
ReLU
(Rectified Linear Unit)

$$f(x) = \max(0,x)$$

- No saturation in the positive region
- Efficient calculation
- In practice, convergence is faster than sigmoid/tanh (about 6 times)
- Downsides
  1. Non-zero average output
  2. And one more problem…

ReLU function (2)

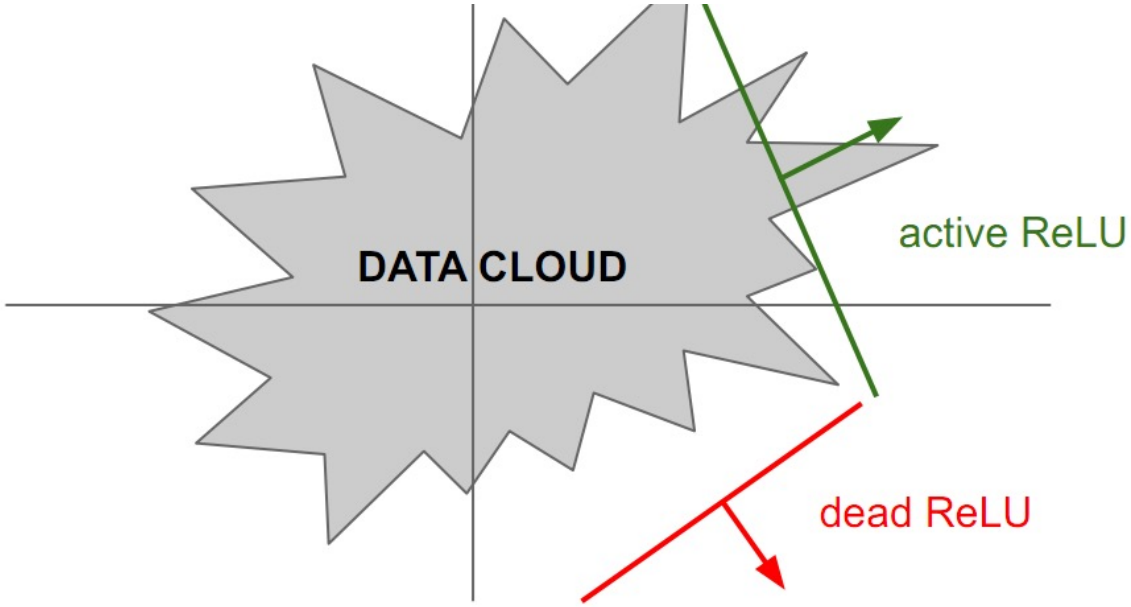




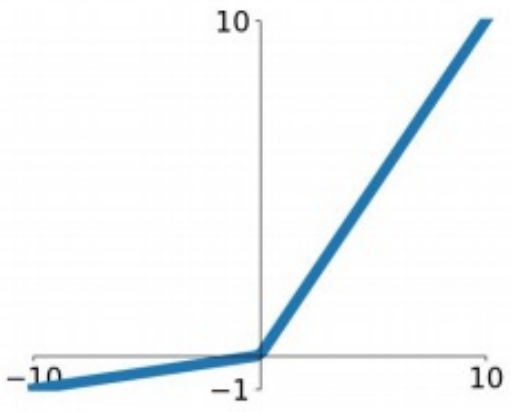
- What happens when x = -10?
- What happens when x = 0?
- What happens when x = 10?

ReLU function (3)

- The ReLU are "thrown" out due to the data set that makes the output being always negative and never updating the weights again.
- Dead ReLU (Dying ReLU)
- Usually initialize the ReLU neuron with a small positive bias (e.g., 0.01)



Leaky ReLU

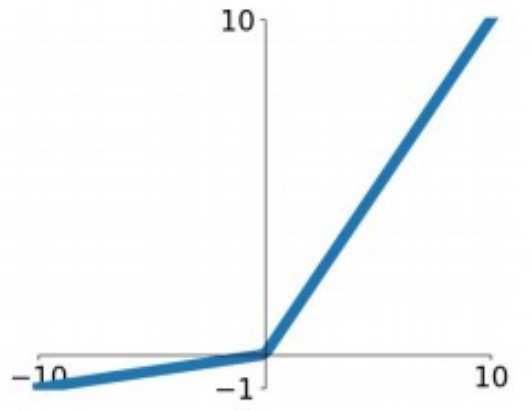


- Does not saturate
- Computationally efficient
- Converges much faster thansigmoid/tanh in practice! (e.g., 6x)
- Will not “die”

Leaky ReLU

$$f(x) = \max(0.01x, x)$$

Parametric Rectifier (PReLU)



- Does not saturate
- Computationally efficient
- Converges much faster thansigmoid/tanh in practice! (e.g., 6x)
- Will not “die”

Leaky ReLU

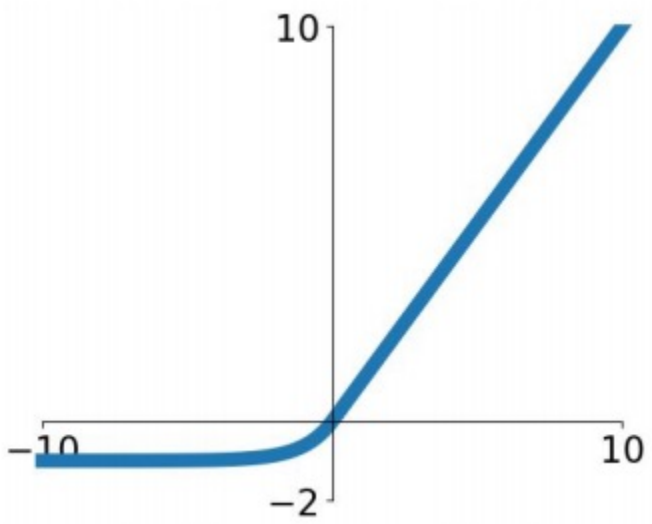
$$f(x) = \max(0.01x, x)$$

Parametric Rectifier (PReLU)

$$f(x) = \max(\alpha x, x)$$

backprop into \alpha (parameter)

Exponential Linear Units



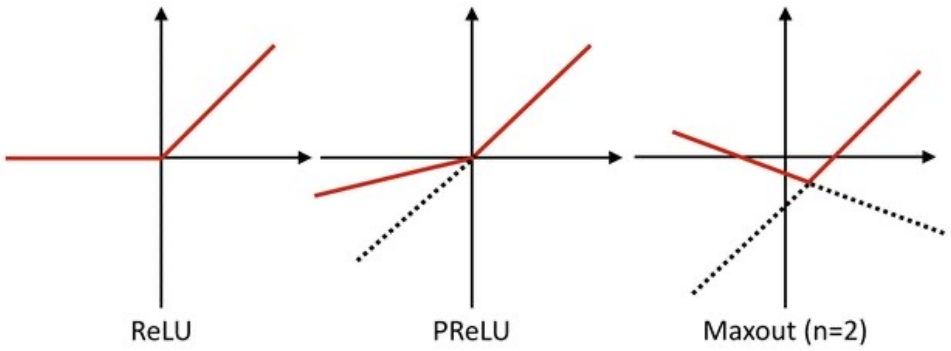
- All benefits of ReLU
- Closer to zero mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise
- Computation requires exp()

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

Maxout function

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

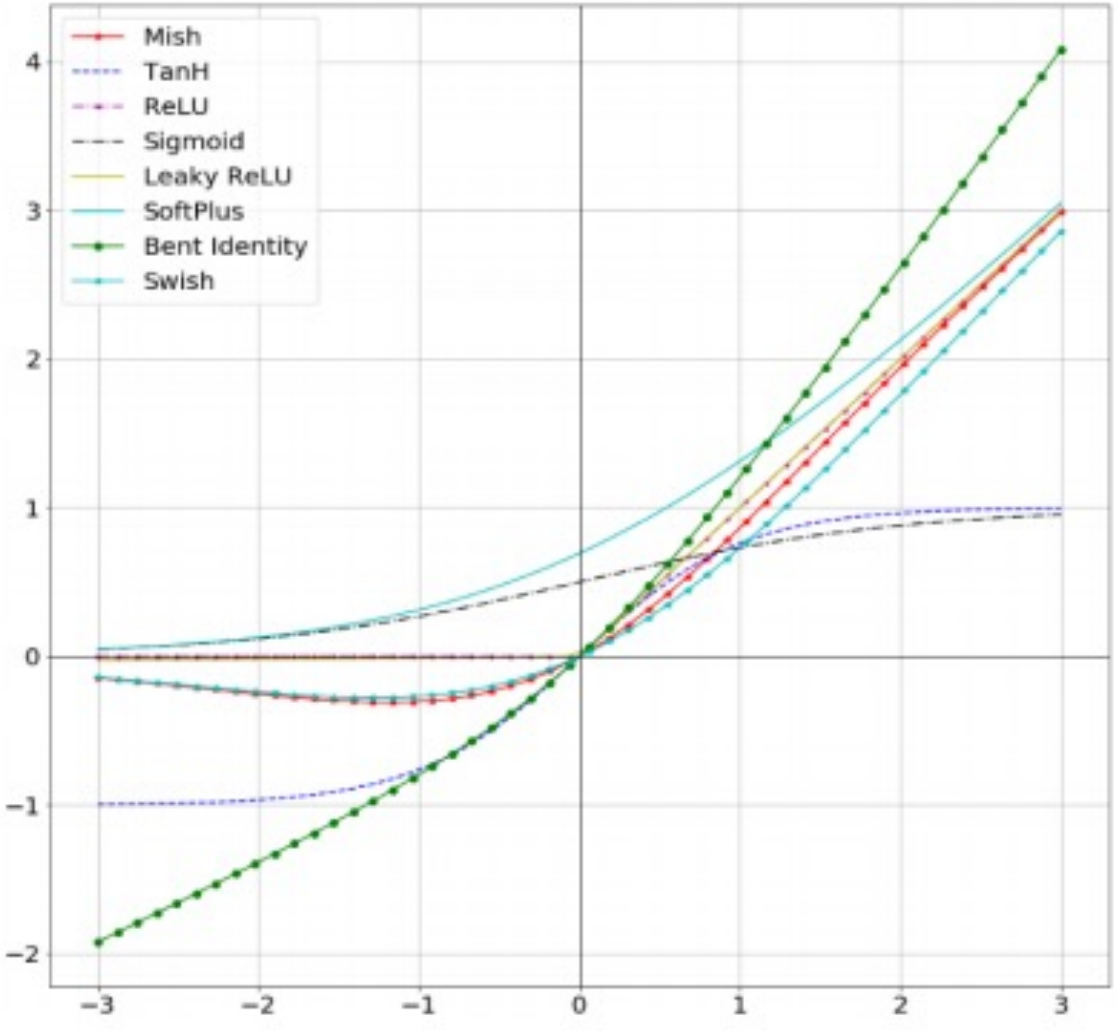
- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!
- Doubles the number of parameters/neuron



Activation function recap

- **In practice**
  - Use ReLU. Be careful with your learning rates
  - Try out Leaky ReLU / Maxout / ELU / SELU
  - To squeeze out some marginal gains
  - Don’t use sigmoid or tanh
- Some new activation functions
  - ReLU6 = min(6, ReLU(x))
  - Swish
  - Mish

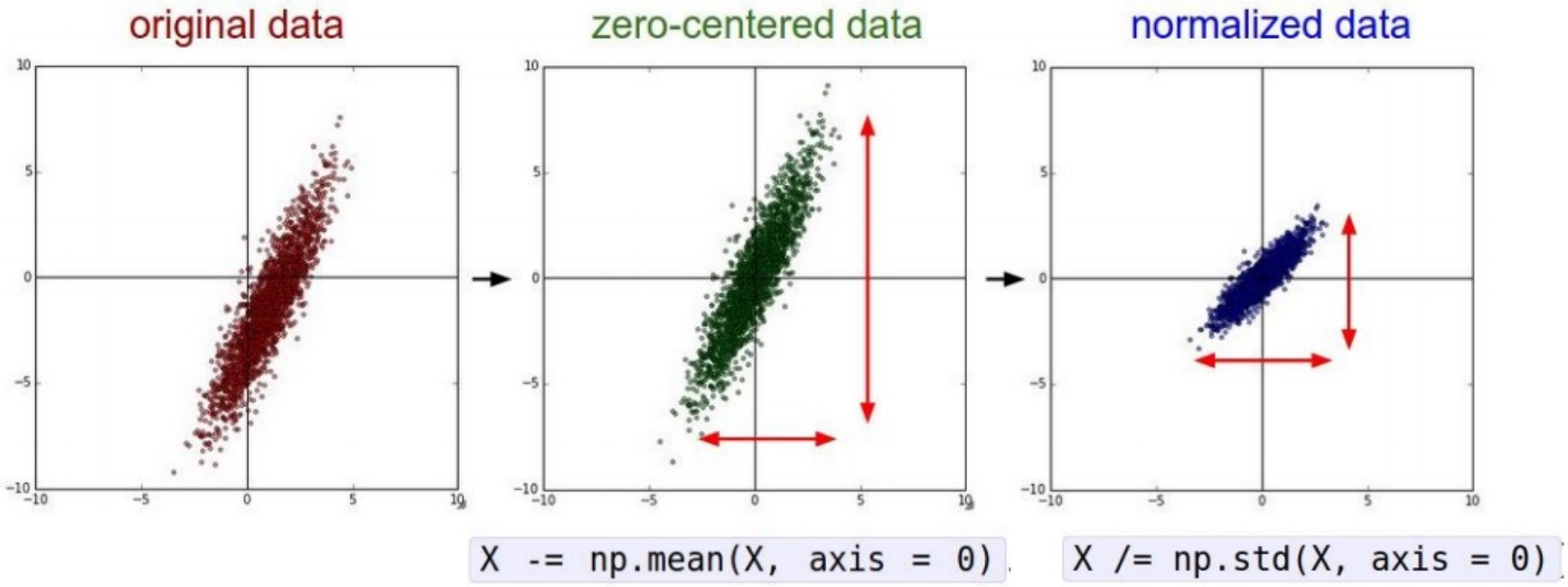
Activation function recap



Data preprocessing

Data preprocessing

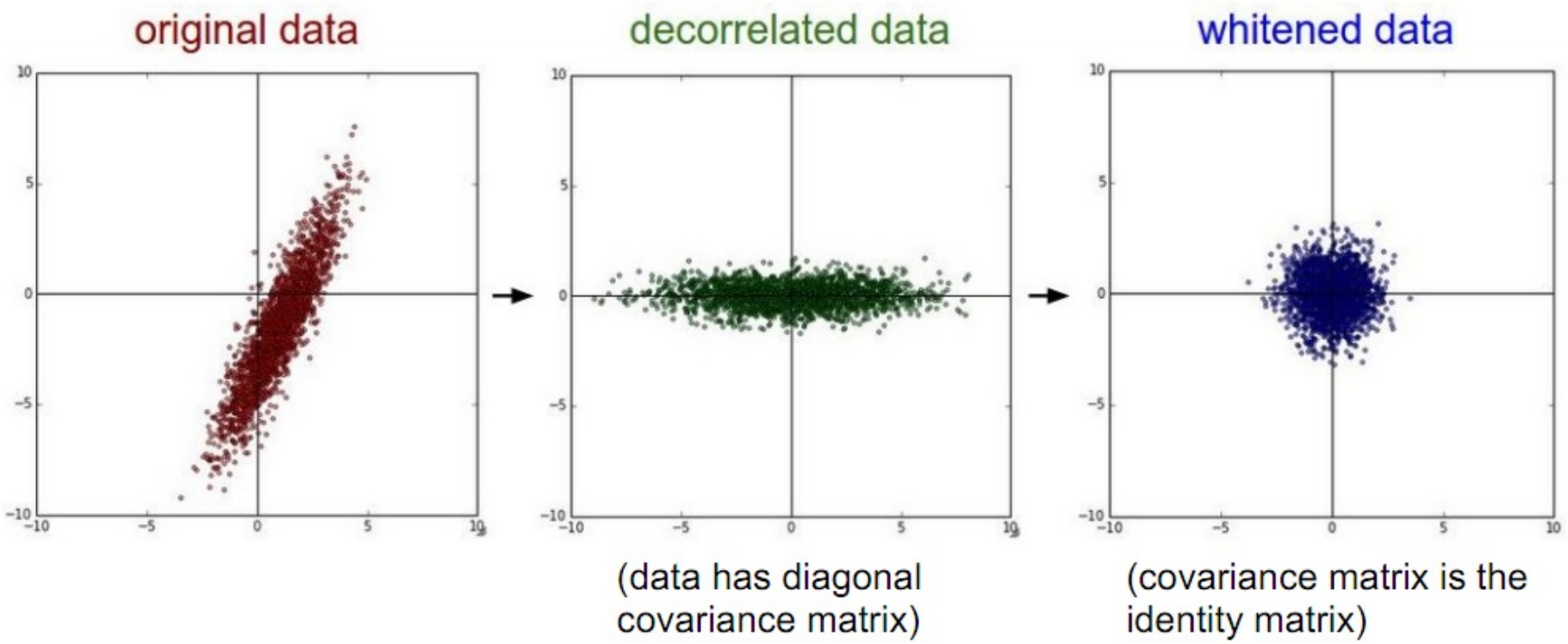
- Transform the data distribution to zero mean: subtract all data samples for the sample mean
- Transform the data distribution to unit standard deviation



(Assume X [NxD] is data matrix, each example in a row)

Data preprocessing (2)

- In reality, we can use PCA or Data Whitening



Data preprocessing (3)

- In practice for images: center only
- Example with CIFAR10 set with 32x32x3 images

- Subtract the mean image (e.g. AlexNet)
  (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)
- Subtract per-channel mean and
  Divide by per-channel std (e.g. ResNet)
  (mean along each channel = 3 numbers)

- Often rarely use PCA or whitening

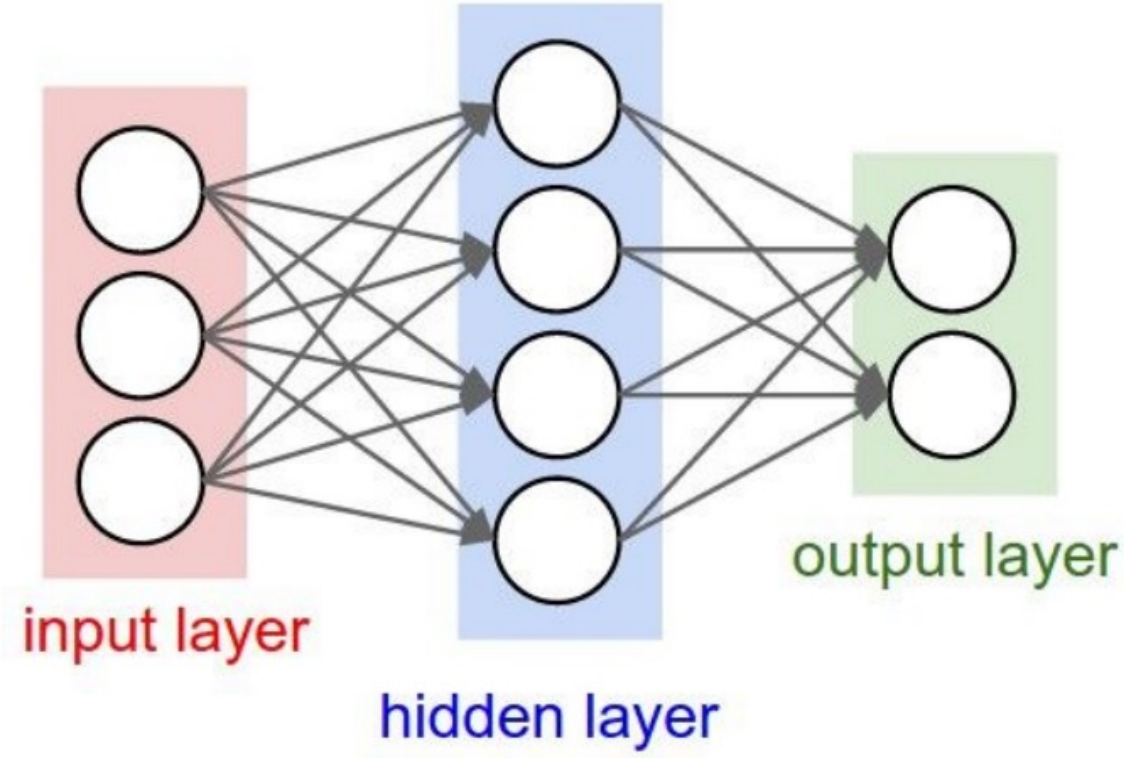
Demo

- https://github.com/trungtv/Preprocessing-for-deep-learning

Weight initializations

Weight initialization

- What if all weights W are initialized to zero?
- Neurons will learn same features in each iterations



Weight initialization (2)

- First idea: Small random numbers (gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01 * np.random.randn(Din, Dout)
```

- Works ~okay for small networks, but problems with deeper networks.

Weight Initialization: Small random numbers

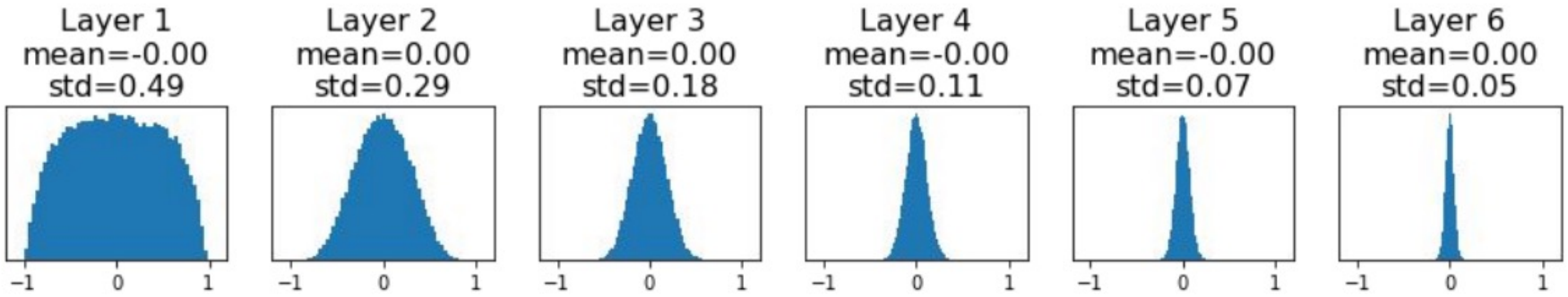
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096

All activations tend to zero for deeper network layers

Q: What do the gradients dL/dW look like?

A: All zero, no learning =(



- Gradient dL/dW = 0
  - w is too small

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

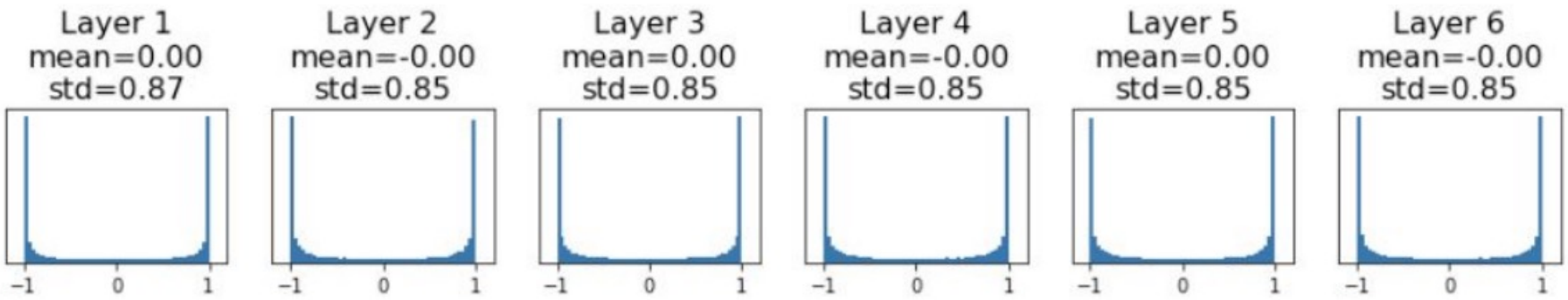
Increase Std

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Increase std of initial weights from 0.01 to 0.05

All activations saturate

Q: What do the gradients look like?



- Gradient dL/dW = 0

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

Xavier initialization

- Assume x and w are iid (independent and identically distributed random variable), and mean 0
- Calculation in forward direction:

$$var(y) = var(w_1x_1 + w_2x_2 + ... + w_{Nin}x_{Nin} + b)$$

$$var(w_ix_i) = E(x_i)^2var(w_i) + E(w_i)^2var(x_i) + var(w_i)var(x_i)$$

$$var(y) = N_{in} * var(w_i) * var(x_i)$$

$$N_{in} * var(w_i) = 1$$

$$var(w_i) = 1/ N_{in}$$

- The same goes for the backward gradient signal stream:

$$var(w_i) = 1/ N_{out}$$

- Average:

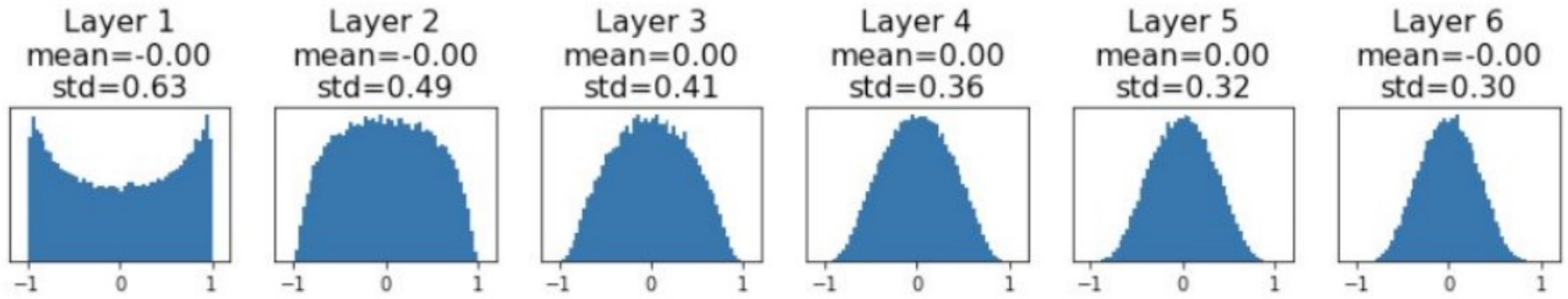
$$var(w_i) = 2/ (N_{in} + N_{out})$$

Xavier initialization (2)

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

"Xavier" initialization:
std = 1/sqrt(Din)

"Just right": Activations are nicely scaled for all layers!



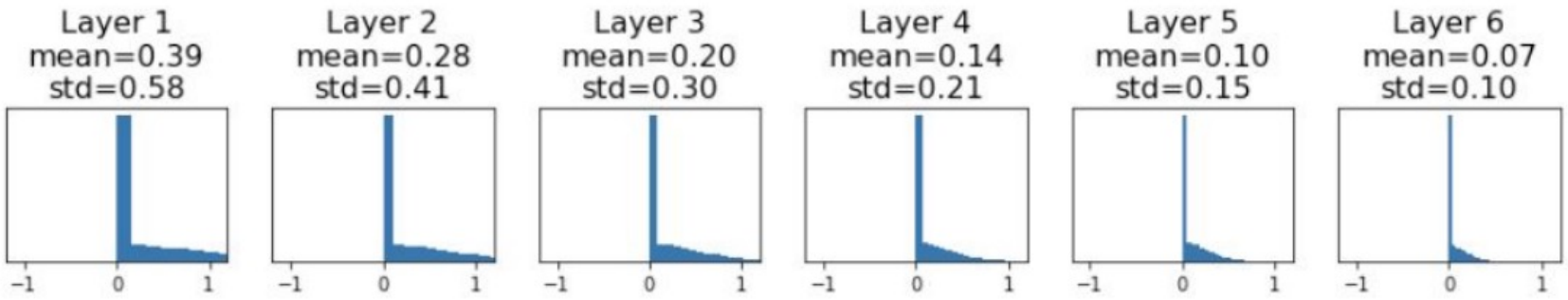
Weight Initialization: Tanh to ReLU

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

Change from tanh to ReLU

Xavier assumes zero centered activation function

Activations collapse to zero again, no learning =(



$$var(y) = var(w_1x_1 + w_2x_2 + ... + w_{Nin}x_{Nin} + b)$$

$$var(y) = N_{in} /2 * var(w_i) * var(x_i)$$

$$N_{in} /2 * var(w_i) = 1$$

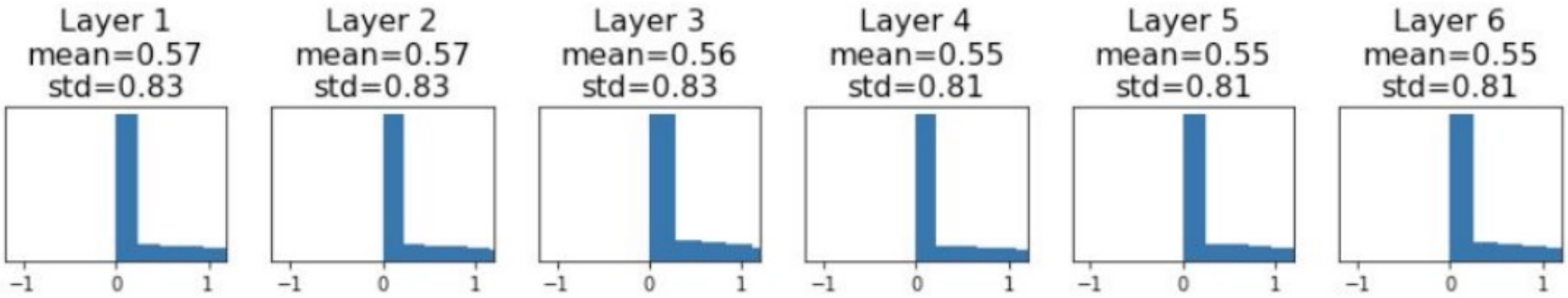
$$var(w_i) = 2/ N_{in}$$

He / MSRA Initialization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) * np.sqrt(2/Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

ReLU correction: std = sqrt(2 / Din)

"Just right": Activations are nicely scaled for all layers!



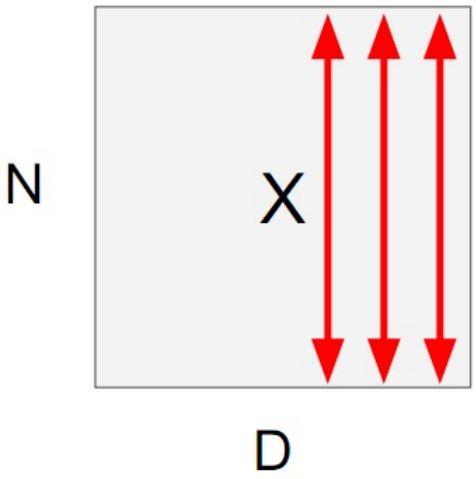
Normalizations

Batch Normalization

- Want the activation function to have an output distribution with zero mean and unit standard deviation? Let's transform with that idea!

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Input: $x : N \times D$



$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$ Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$ Per-channel var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$ Normalized x, Shape is N x D

Batch Normalization

- The ideal of zero expectation unit standard deviation is too tight! Can cause the model to be underfitting.
  - Loosen the model, create an exit for the model if it does not want to be bound.

Input: $x : N \times D$

Learnable scale and shift parameters:

$\gamma, \beta : D$

Learning $\gamma = \sigma$, $\beta = \mu$ will recover the identity function!

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$ Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$ Per-channel var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$ Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$ Output, Shape is N x D

Batch Normalization: Test-time

- Estimates depend on minibatch; can't do this at test-time!

Input: $x : N \times D$

Learnable scale and shift parameters:

$$\gamma, \beta : D$$

Learning $\gamma = \sigma$, $\beta = \mu$ will recover the identity function!

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$ Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$ Per-channel var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$ Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$ Output, Shape is N x D

Batch Normalization: Test-time (2)

Input: $x : N \times D$

Learnable scale and shift parameters:

$\gamma, \beta : D$

During testing batchnorm becomes a linear operator!
Can be fused with the previous fully-connected or conv layer

$\mu_j =$ (Running) average of values seen during training — Per-channel mean, shape is D

$\sigma_j^2 =$ (Running) average of values seen during training — Per-channel var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$ — Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$ — Output, Shape is N x D

Batch Normalization

Batch Normalization for **fully-connected** networks

$x: N \times D$

Normalize ↓

$\mu, \sigma: 1 \times D$

$\gamma, \beta: 1 \times D$

$y = \gamma(x-\mu)/\sigma+\beta$

Batch Normalization for **convolutional** networks (Spatial Batchnorm, BatchNorm2D)

$x: N \times C \times H \times W$

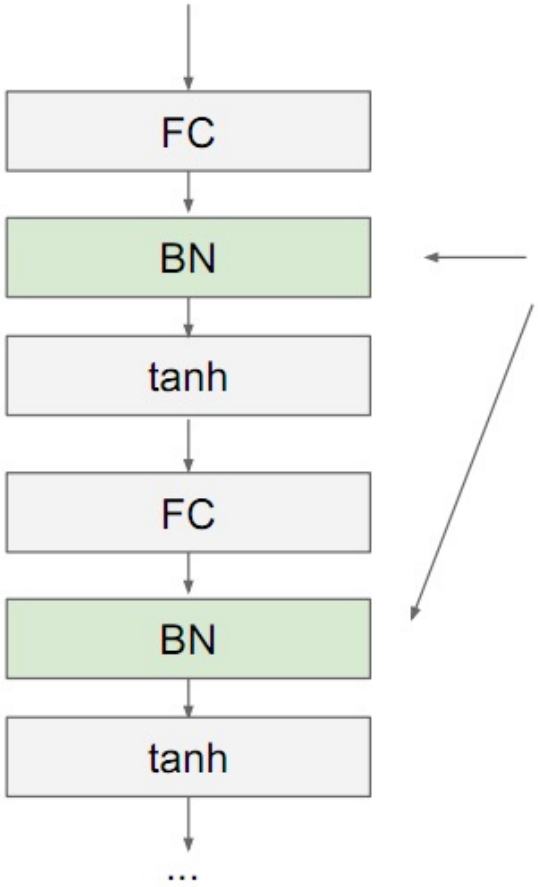
Normalize ↓ ↓ ↓

$\mu, \sigma: 1 \times C \times 1 \times 1$

$\gamma, \beta: 1 \times C \times 1 \times 1$

$y = \gamma(x-\mu)/\sigma+\beta$

Batch Normalization: Usage



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

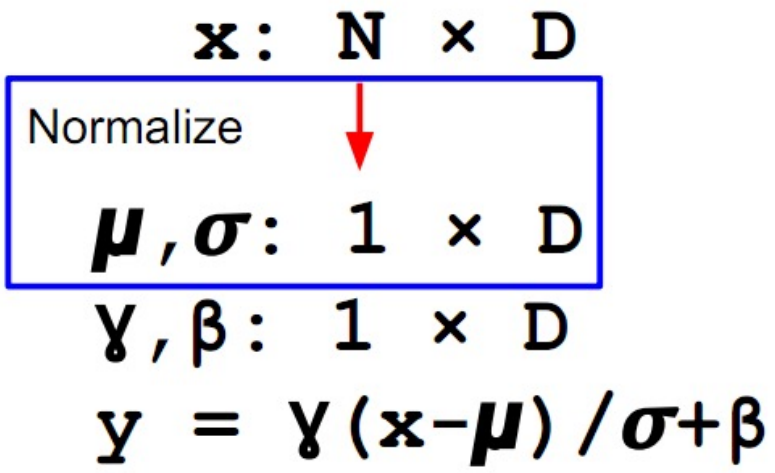
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Batch Normalization: Pros & Cons

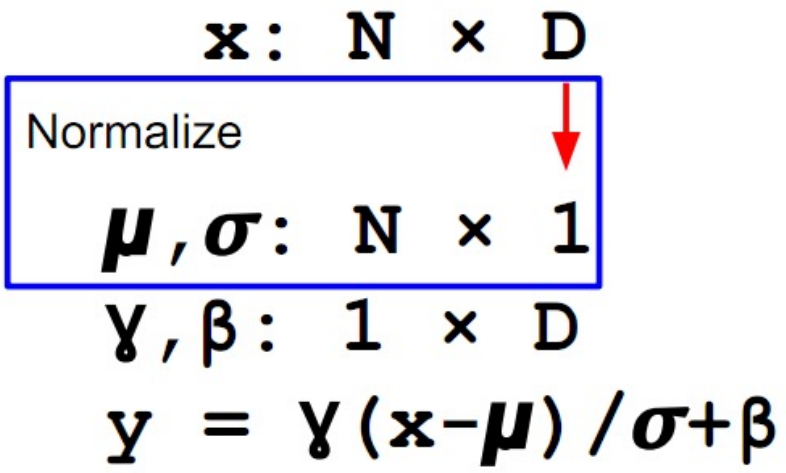
- Makes deep networks much easier to train!
- Improves gradient flow
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!
- Behaves differently during training and testing: this a very common source of bugs!

Layer Normalization

Batch Normalization for fully-connected networks

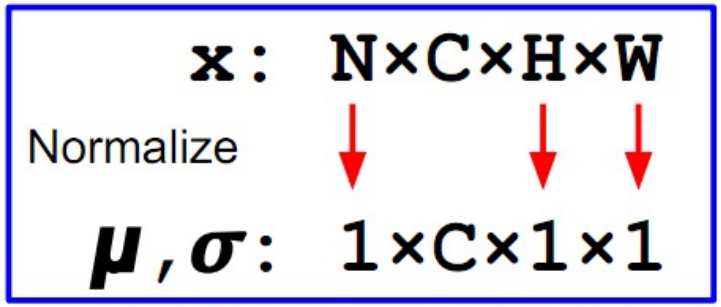


Layer Normalization for fully-connected networks
Same behavior at train and test!
Can be used in recurrent networks

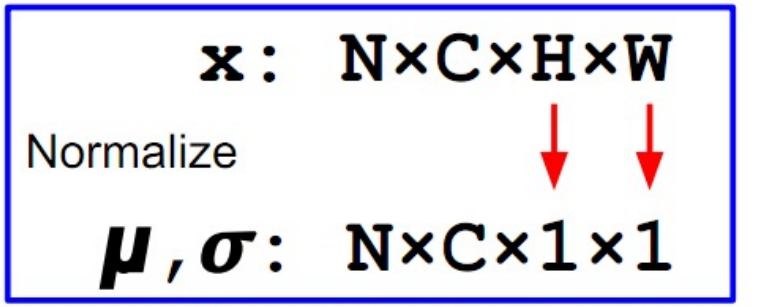


Instance Normalization

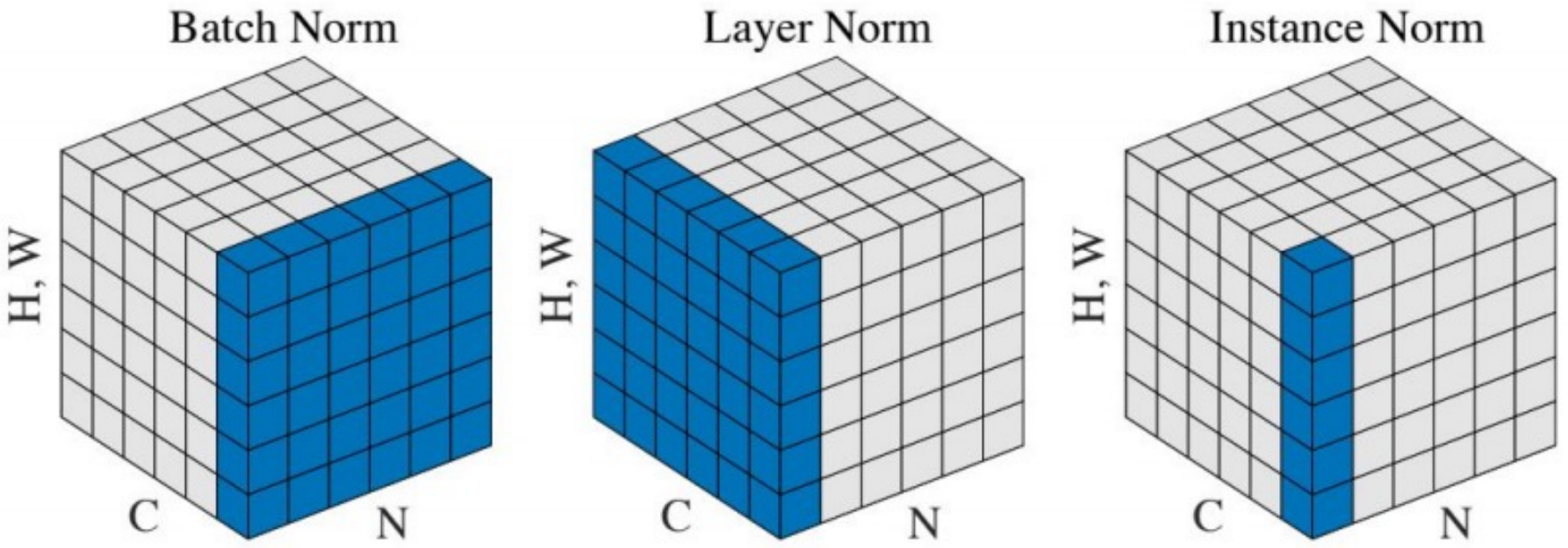
Batch Normalization for convolutional networks



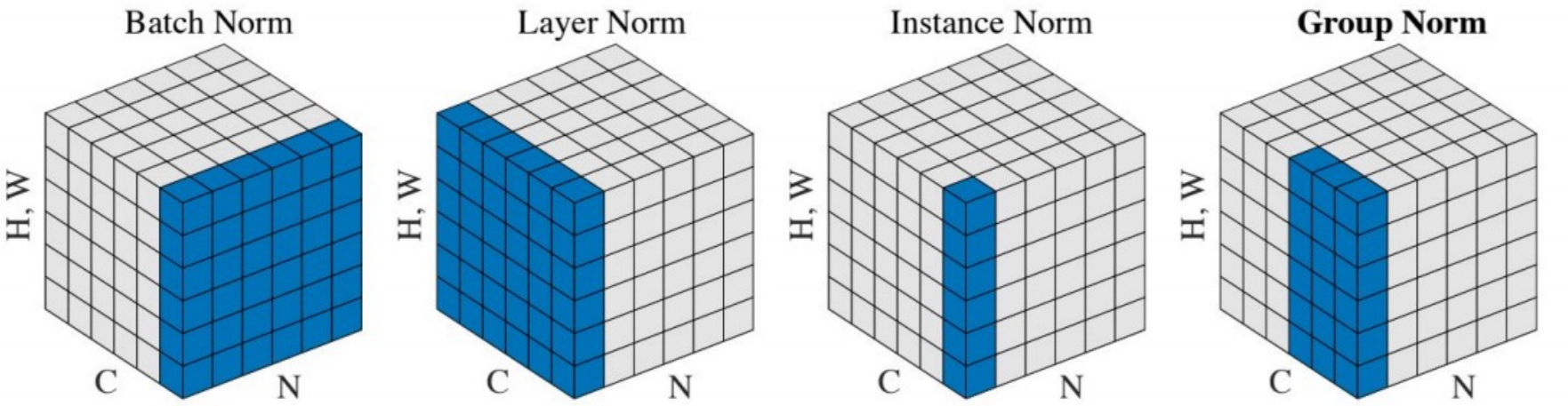
Instance Normalization for convolutional networks
Same behavior at train / test!



Comparison of Normalization Layers



Group Normalization



References

1. http://cs231n.stanford.edu
2. https://prateekvjoshi.com/2016/03/29/understanding-xavier-initialization-in-deep-neural-networks/

ĐẠI HỌC BÁCH KHOA

25 YEARS ANNIVERSARY SOICT

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for your attention!!!

soict.hust.edu.vn/ fb.com/groups/soict