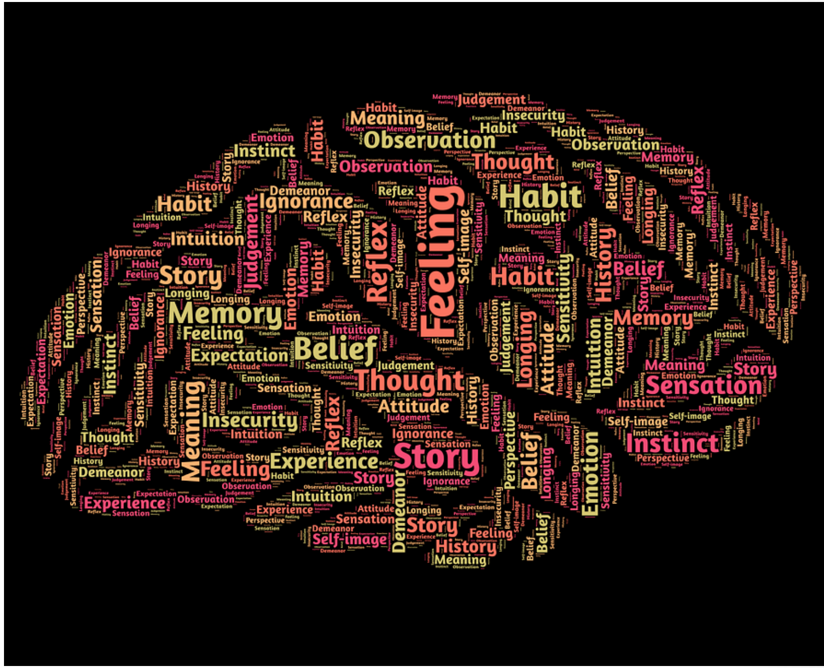
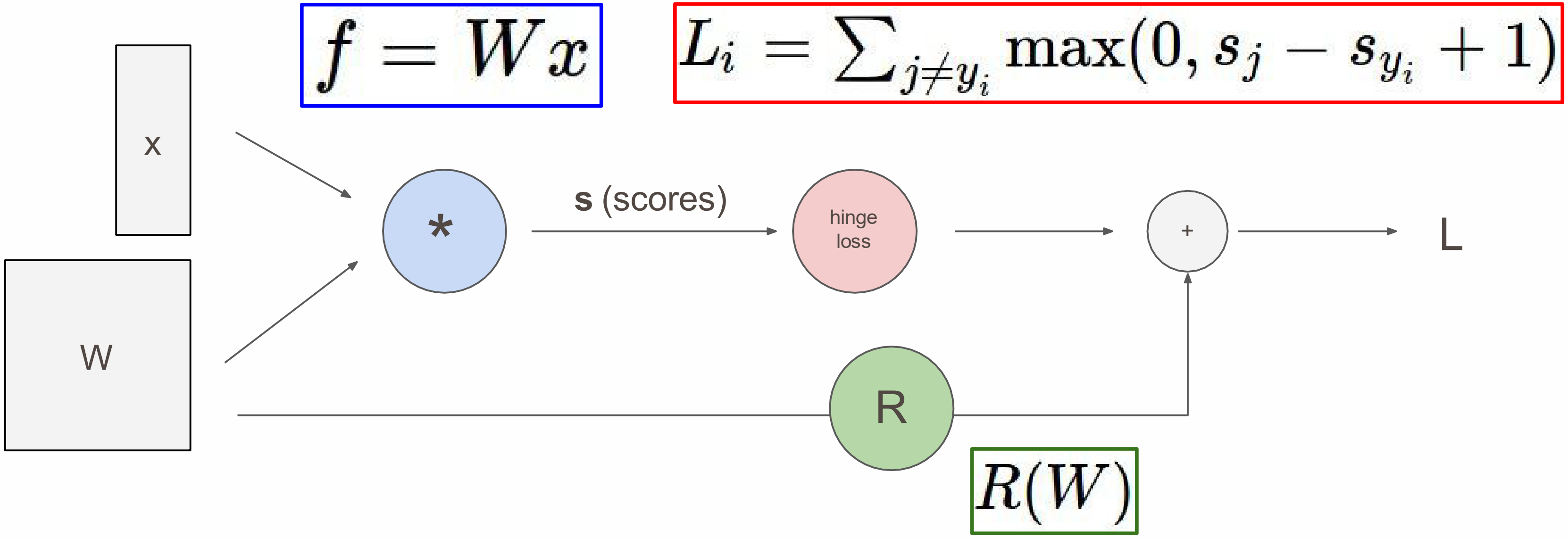
ACTIVATION FUNCTION AND NORMALIZATION

Chih-Chung Hsu (許志仲)
Institute of Data Science
National Cheng Kung University
https://cchsu.info



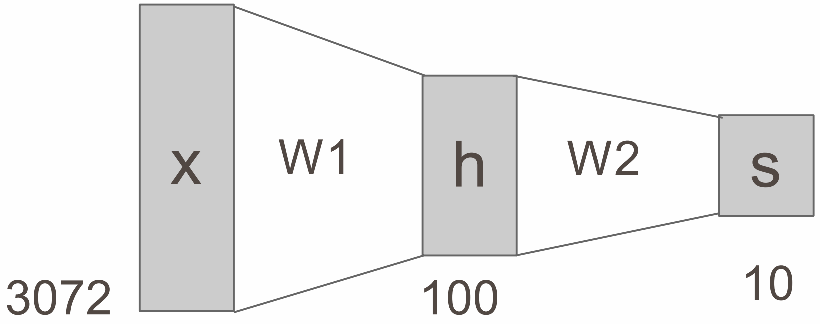
Recap: Computational graphs

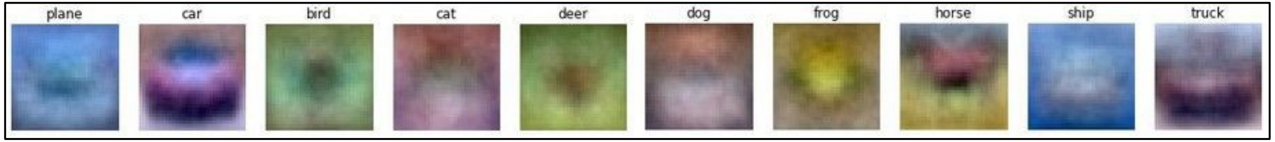


Recap: Neural Networks

Linear score function: $f = Wx$

2-layer Neural Network $f = W_2 \max(0, W_1 x)$





Recap: Convolutional Neural Networks

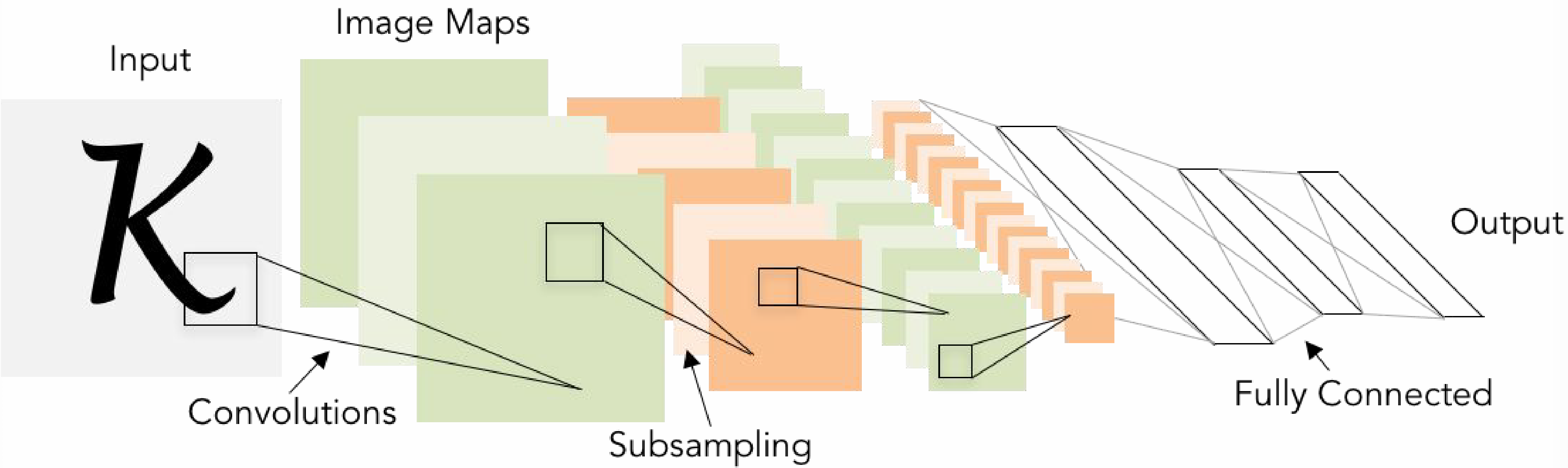
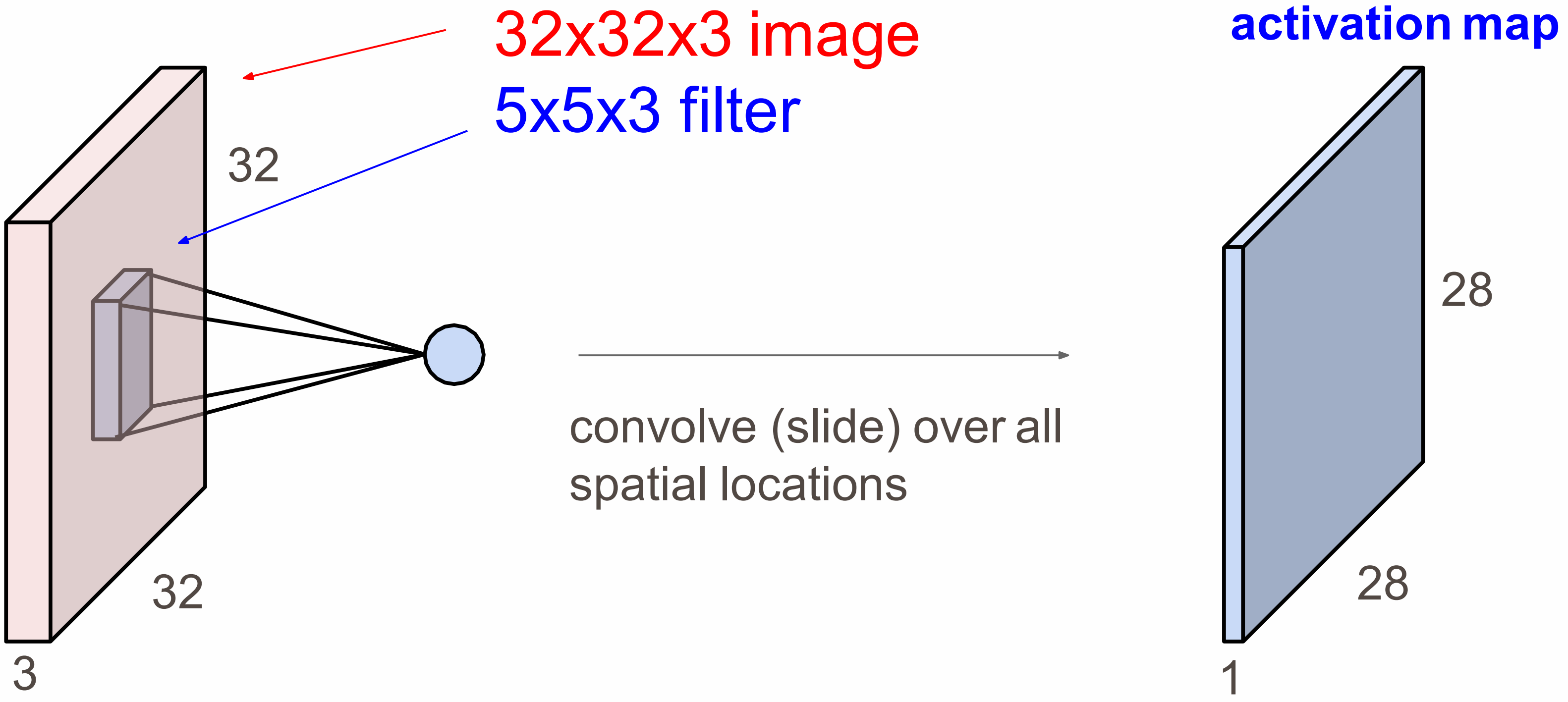


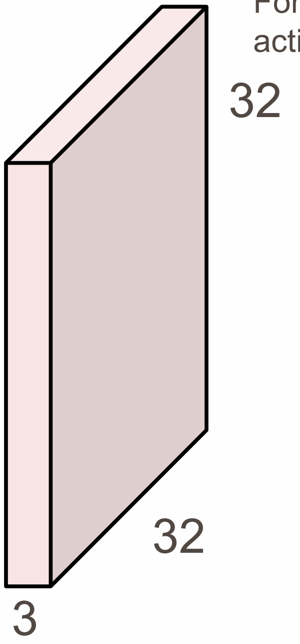
Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1

Recap: Convolutional Layer



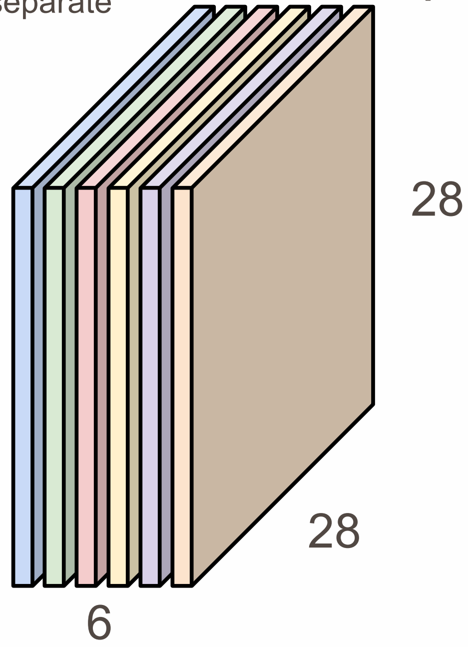
Recap: Convolutional Layer

For example, if we had 6 5x5 filters, we'll get 6 separate activation maps



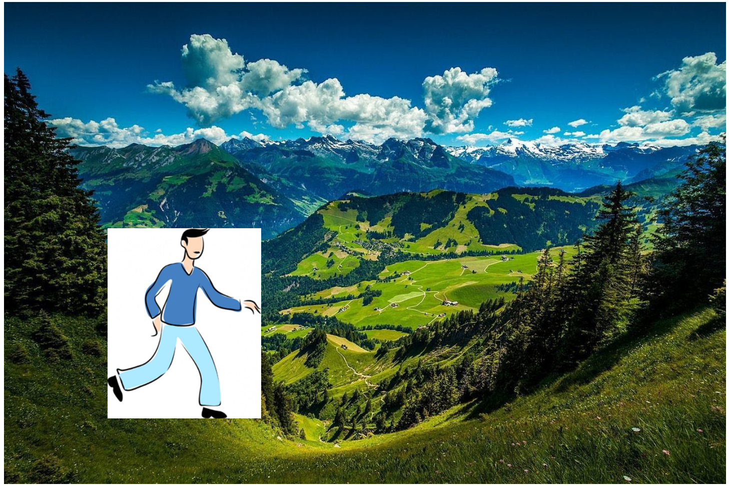
Convolution Layer

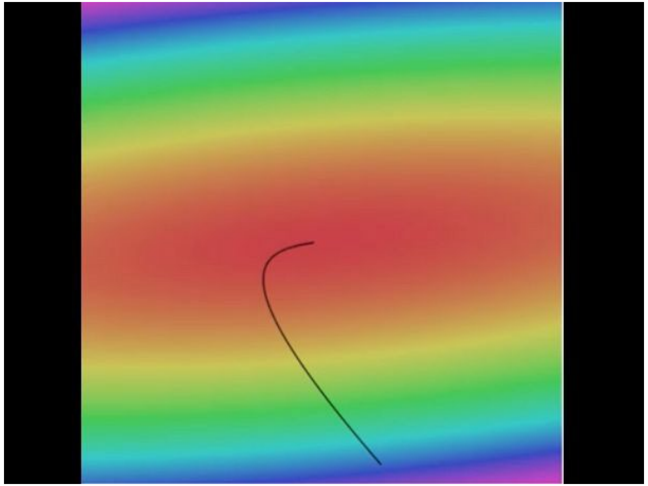
activation maps



We stack these up to get a "new image" of size 28x28x6!

Recap: Learning network parameters through optimization





```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```


Slide Credit cs231n

Recap: Mini-batch SGD

- Loop:
- Sample a batch of data
- Forward prop it through the graph (network), get loss
- Backprop to calculate the gradients
- Update the parameters using the gradient

Overview

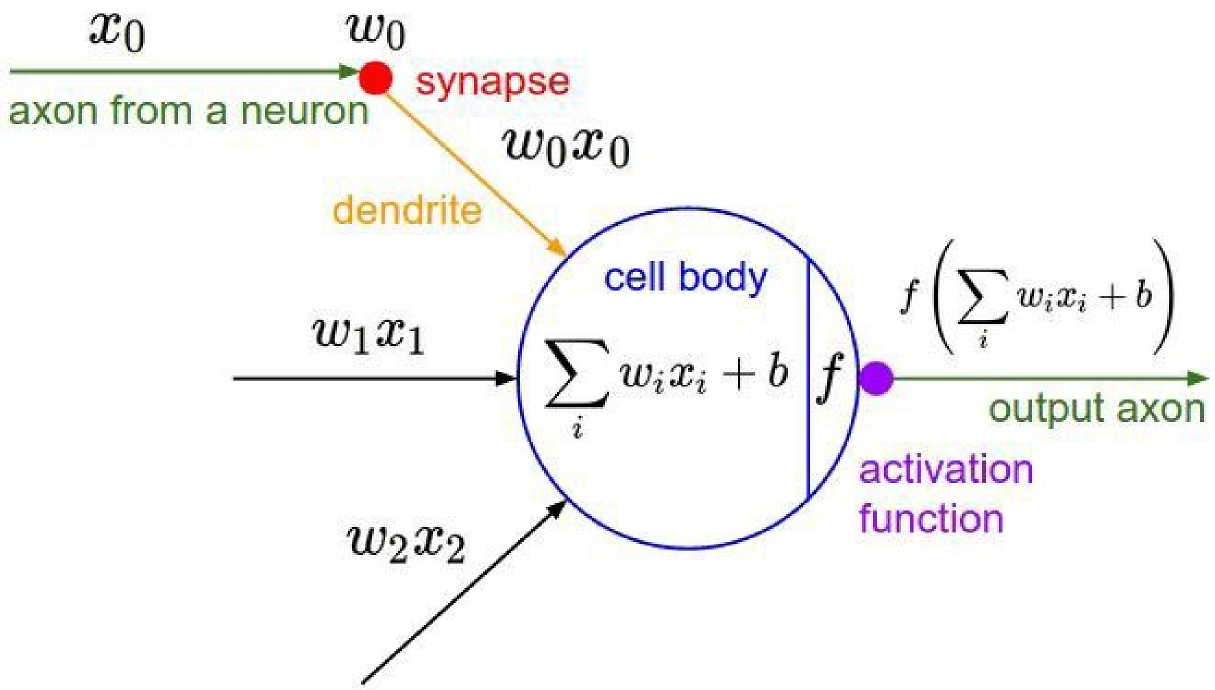
- One time setup
  - activation functions, preprocessing, weight initialization, regularization, gradient checking
- Training dynamics
  - babysitting the learning process,
  - parameter updates, hyperparameter optimization
- Evaluation
  - model ensembles, test-time augmentation

Part 1

- Activation Functions
- Data Preprocessing
- Weight Initialization
- Batch Normalization
- Babysitting the Learning Process
- Hyperparameter Optimization

ACTIVATION FUNCTIONS

Activation Functions

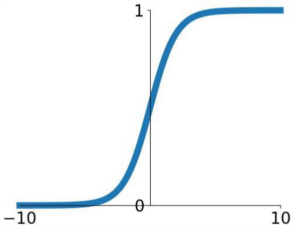


Slide Credit cs231n

Activation Functions

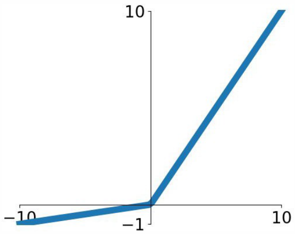
Sigmoid

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



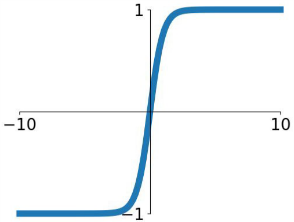
Leaky ReLU

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



tanh

$$\tanh(x)$$

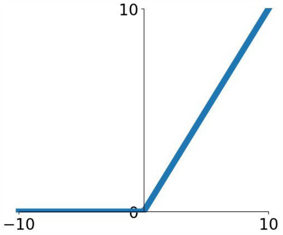


Maxout

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

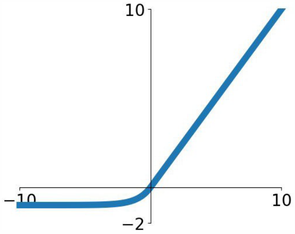
ReLU

$$\max(0, x)$$



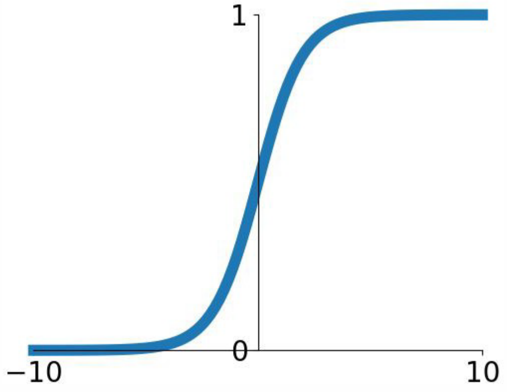
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Slide Credit cs231n

Activation Functions



Sigmoid

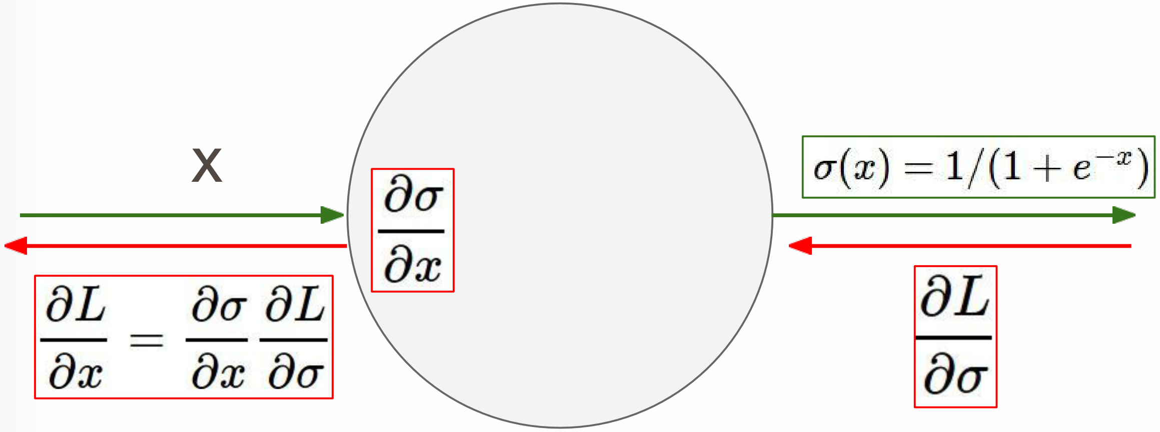
$$\sigma(x) = 1/(1+e^{-x})$$

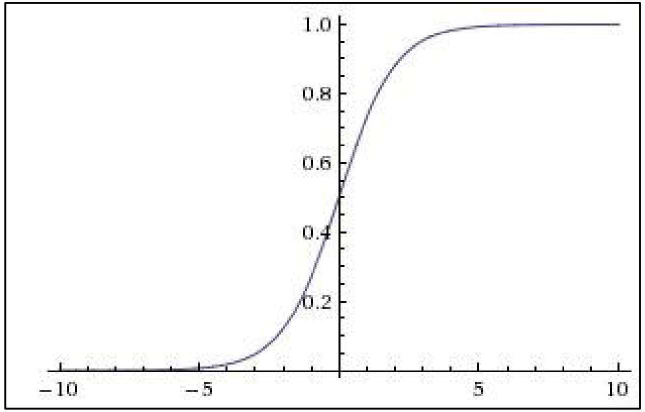
Squashes numbers to range [0,1]
Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

problems:

1. Saturated neurons "kill" the gradients

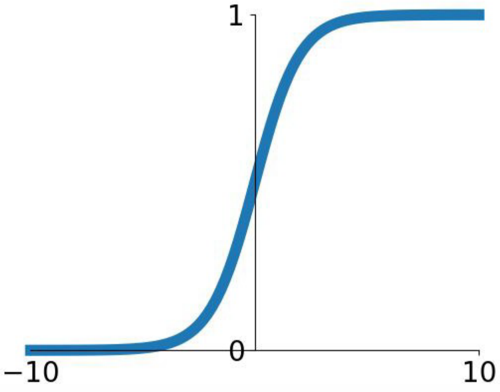
sigmoid gate





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

Activation Functions



Sigmoid

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

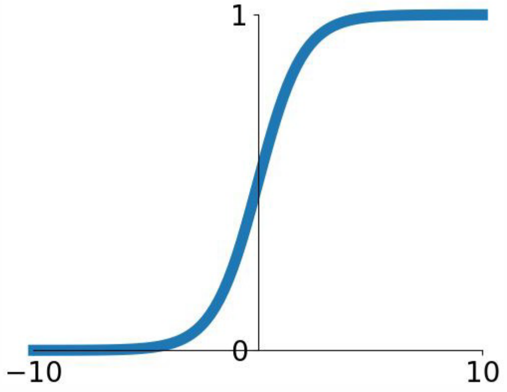
Squashes numbers to range [0,1]
Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

problems:

1. Saturated neurons "kill" the gradients
2. Sigmoid outputs are not zero-centered

Slide Credit cs231n

Activation Functions



Sigmoid

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

Squashes numbers to range [0,1]
Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

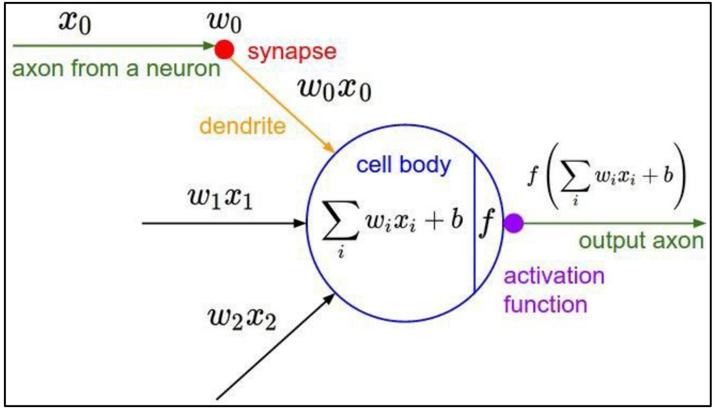
problems:

1. Saturated neurons "kill" the gradients
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive

Slide Credit cs231n

Consider what happens when the input to a neuron is always positive...

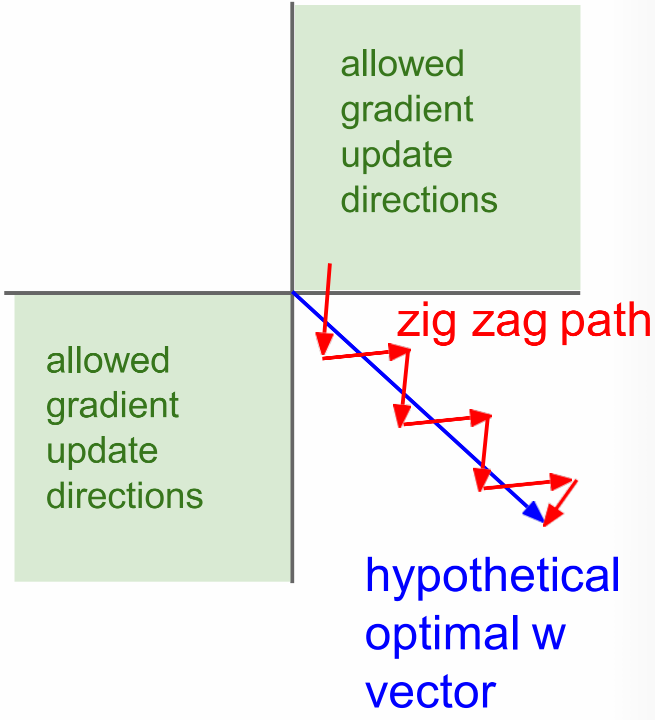
$$f\left(\sum_i w_i x_i + b\right)$$



What can we say about the gradients on **w**?

Consider what happens when the input to a neuron is always positive...

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



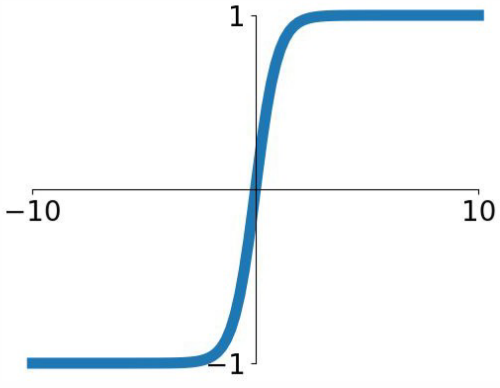
What can we say about the gradients on **w**?

Always all positive or all negative :(

(For a single element! Minibatches help)

Slide Credit cs231n

Activation Functions

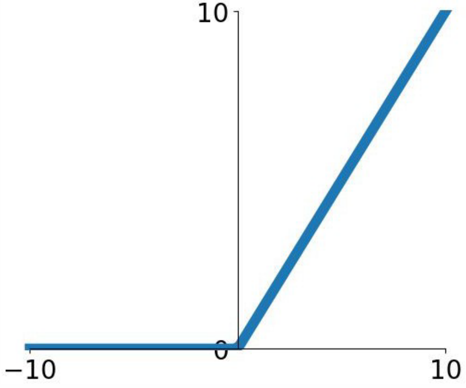


tanh(x)

- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

[LeCun et al., 1991]

Activation Functions



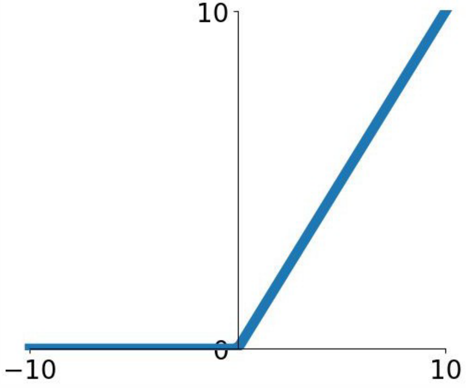
Computes f(x) = max(0,x)

ReLU
(Rectified Linear Unit)

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

[Krizhevsky et al., 2012]

Activation Functions



Computes f(x) = max(0,x)

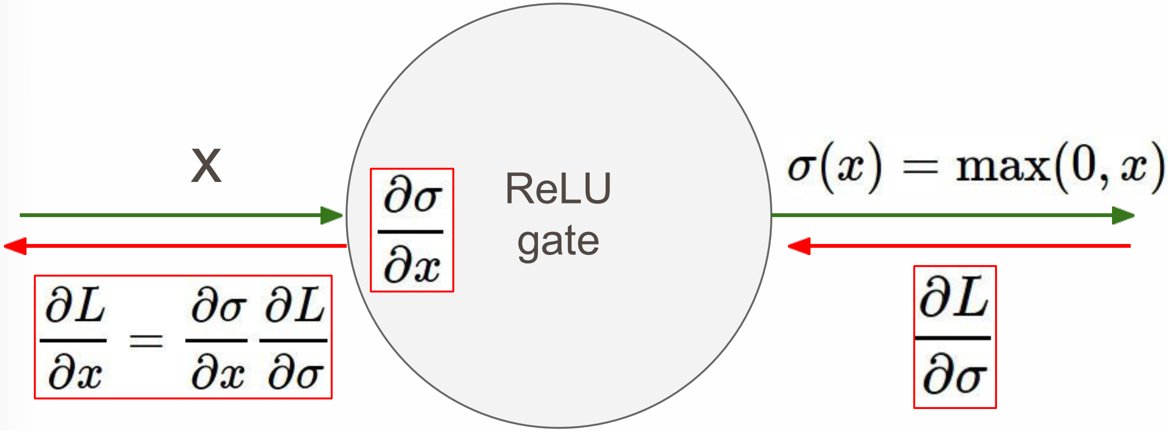
ReLU
(Rectified Linear Unit)

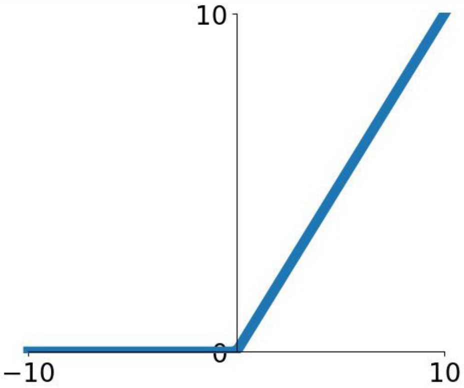
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

- Not zero-centered output
- An annoyance:

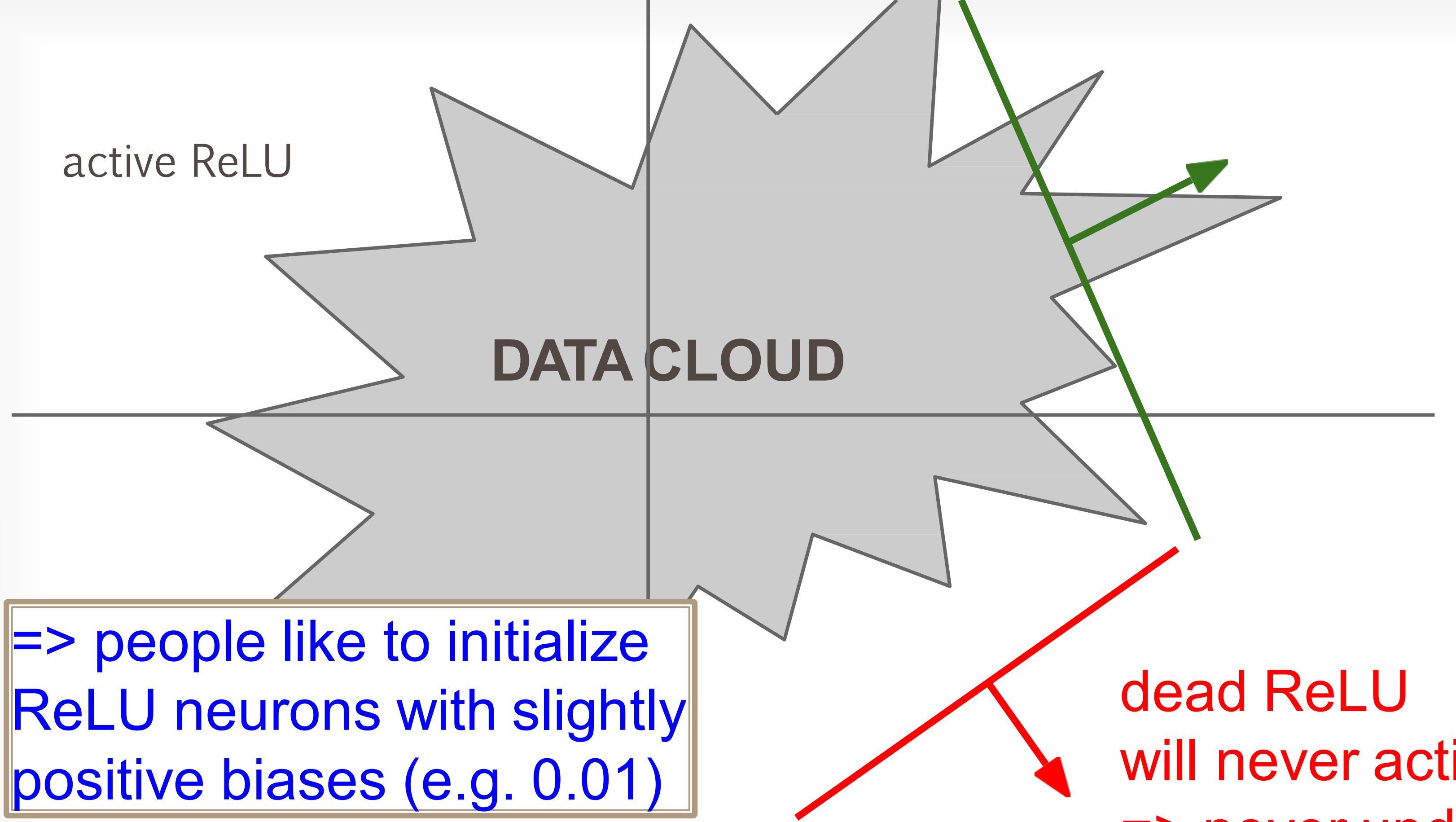
hint: what is the gradient when x < 0?

[Krizhevsky et al., 2012]





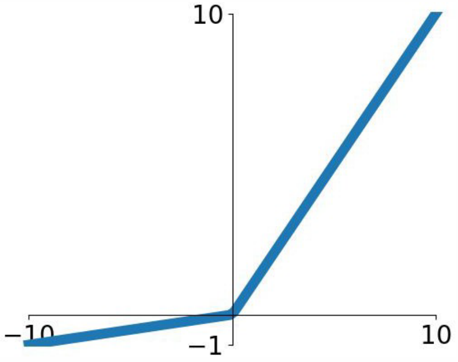
What happens when x = -10?
What happens when x = 0?
What happens when x = 10?



=> people like to initialize ReLU neurons with slightly positive biases (e.g. 0.01)

dead ReLU
will never activate
=> never update

Activation Functions



- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not "die".**

Leaky ReLU

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

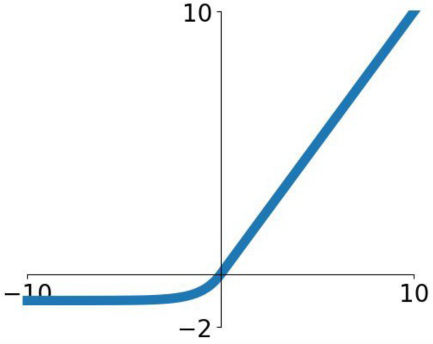
[Mass et al., 2013] [He et al., 2015]

Parametric Rectifier (PReLU)

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

backprop into \alpha (parameter)

Activation Functions



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

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

- Computation requires exp()

Exponential Linear Units (ELU)

[Clevert et al., 2015]

Slide Credit cs231n

Maxout "Neuron"

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

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

Problem: doubles the number of parameters/neuron :(

[Goodfellow et al., 2013]

TLDR: In practice:

- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU
- Try out tanh but don't expect much
- Don't use sigmoid

SOTA Activation Function so far

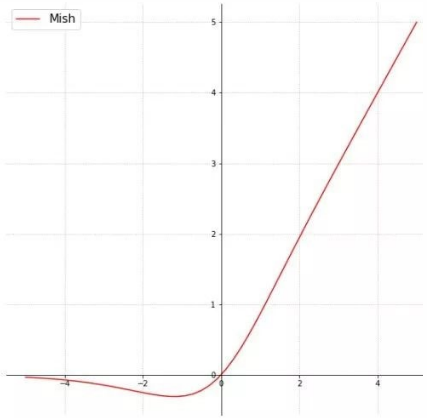
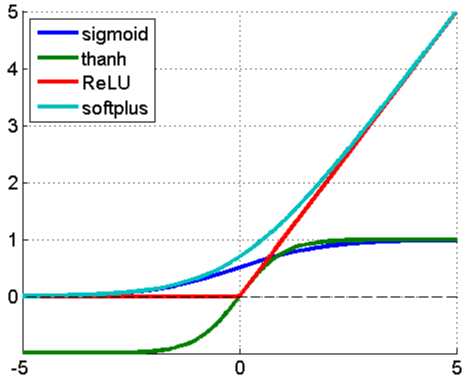


Figure 1. Mish Activation Function



$$f(x) = x\tanh(softplus(x)) = x\tanh(\ln(1+e^{x}))$$

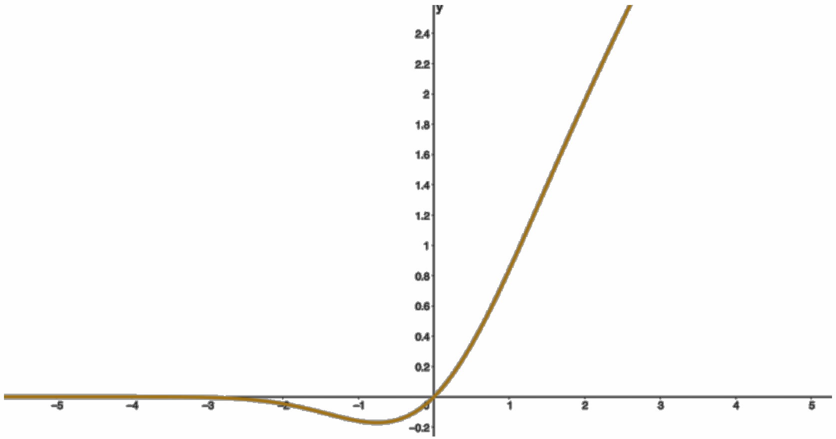
MISH

$$f(x) = x \cdot \text{sigmoid}(\beta x)$$

SWISH

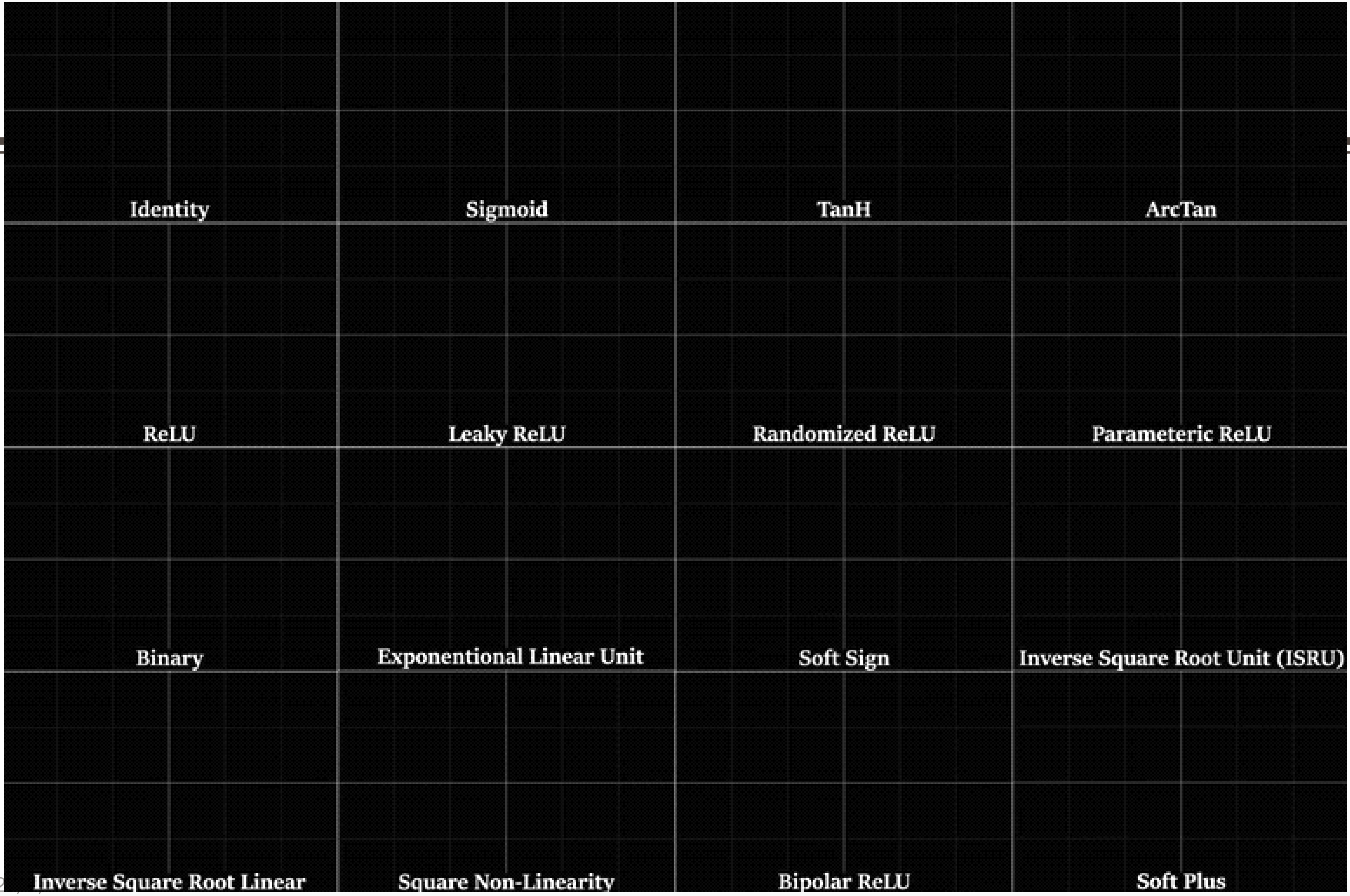
Slide Credit cs231n

SOTA Activation Function so far



$$\mathbf{GELU}(x) = 0.5x\left(1 + \tanh\left(\sqrt{2/\pi}(x + 0.044715x^3)\right)\right)$$

GELU

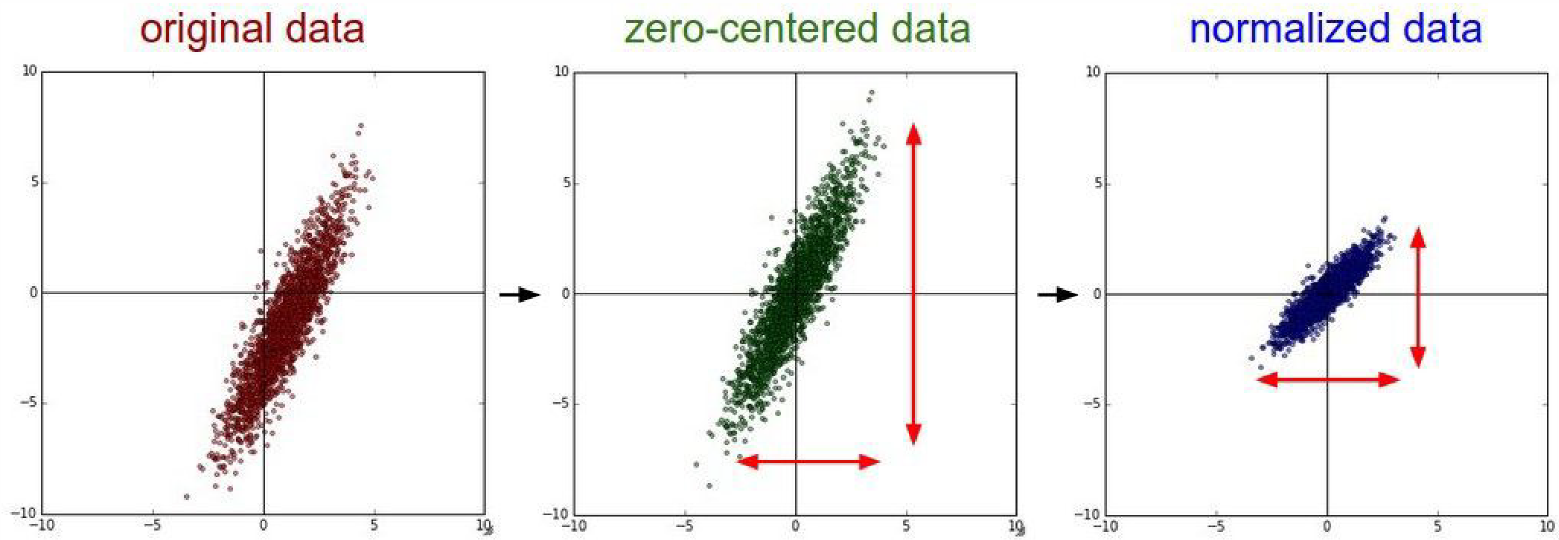


Slide Credit cs231n

Source:https://mlfromscratch.com/activation-functions-explained/#/

DATA PREPROCESSING

Data Preprocessing




```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```

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

Slide Credit cs231n

Remember: Consider what happens when

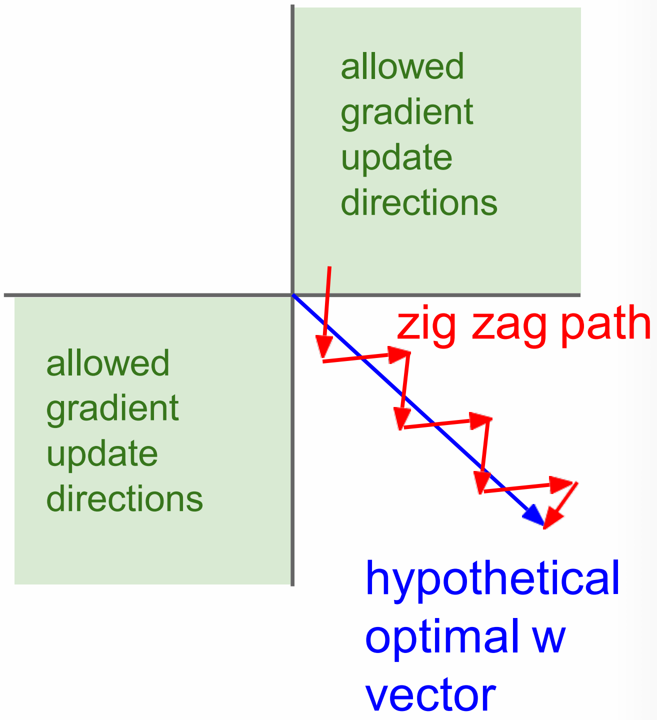
$$f\left(\sum_i w_i x_i + b\right)$$

the input to a neuron is always positive...

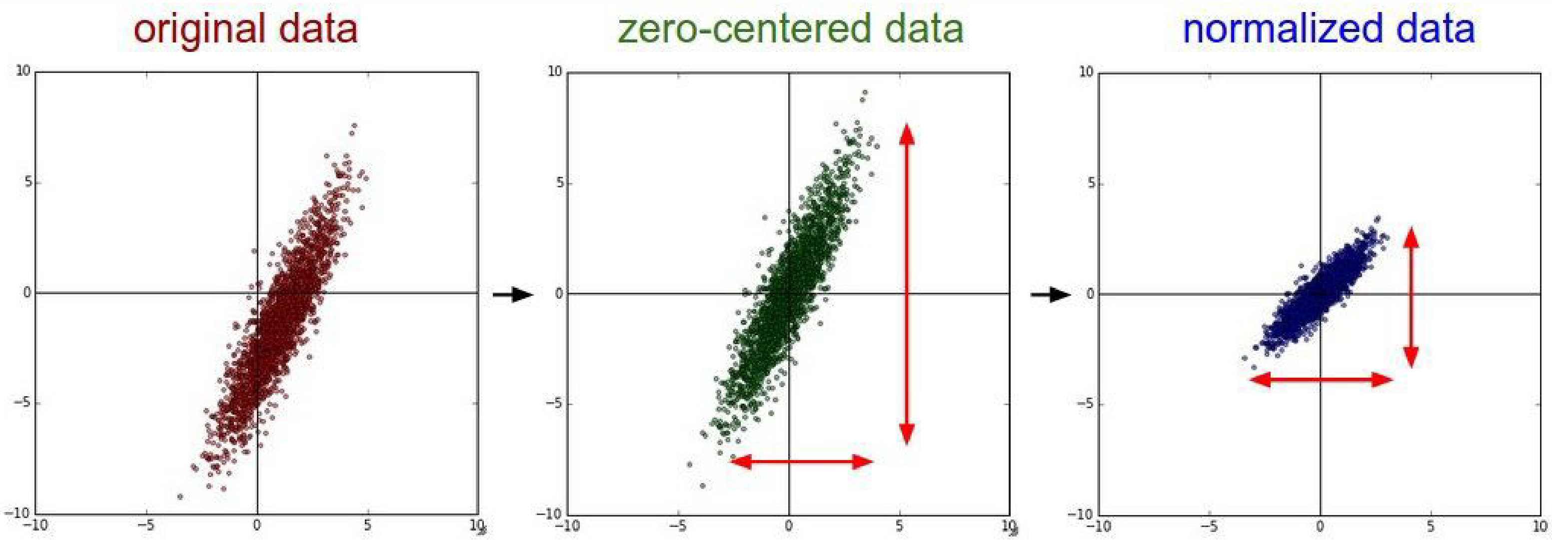
What can we say about the gradients on **w**?

Always all positive or all negative :(

(this is also why you want zero-mean data!)



Data Preprocessing



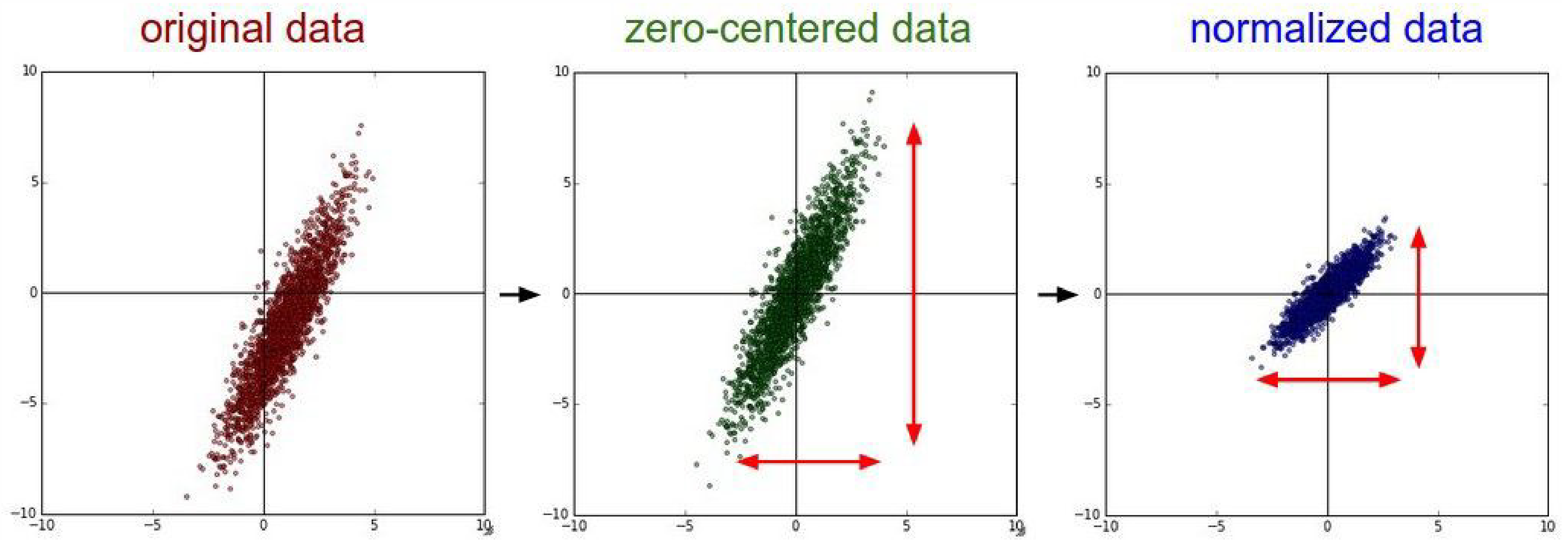

```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```

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

Slide Credit cs231n

Data Preprocessing



(data has diagonal covariance matrix)

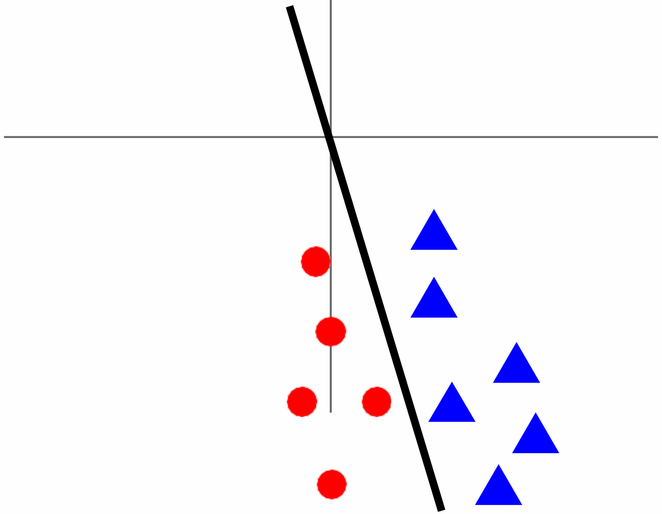
(covariance matrix is the identity matrix)

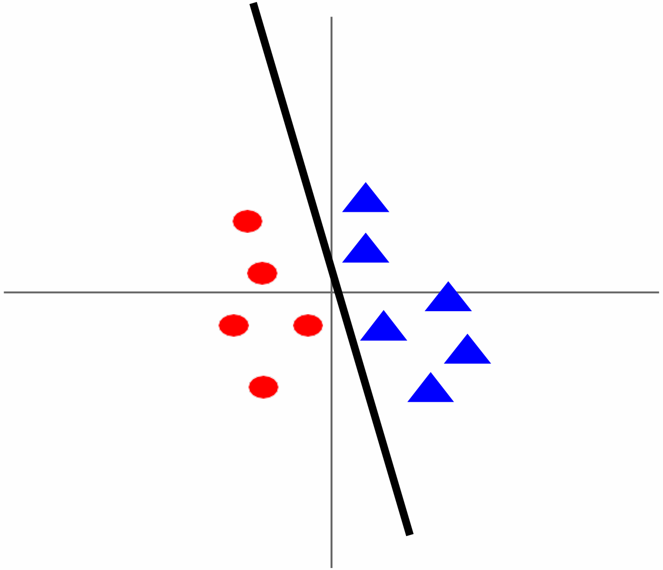
In practice, you may also see **PCA** and **Whitening** of the data

Slide Credit cs231n

Data Preprocessing

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize





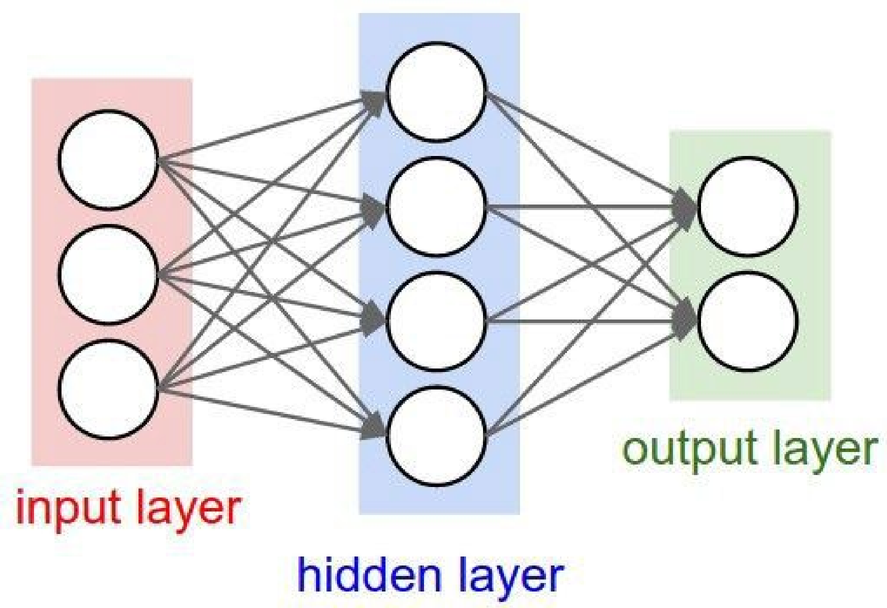
After normalization: less sensitive to small changes in weights; easier to optimize

In practice for Images: center only

- e.g. consider CIFAR-10 example with [32,32,3] 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)
- Not common to do PCA or whitening

WEIGHT INITIALIZATION

Q: what happens when W=constant init is used?



Slide Credit cs231n

First idea: Small random numbers

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

(gaussian with zero mean and 1e-2 standard deviation)

First idea: Small random numbers

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

(gaussian with zero mean and 1e-2 standard deviation)

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

Weight Initialization: Activation statistics

```
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

Weight Initialization: Activation statistics

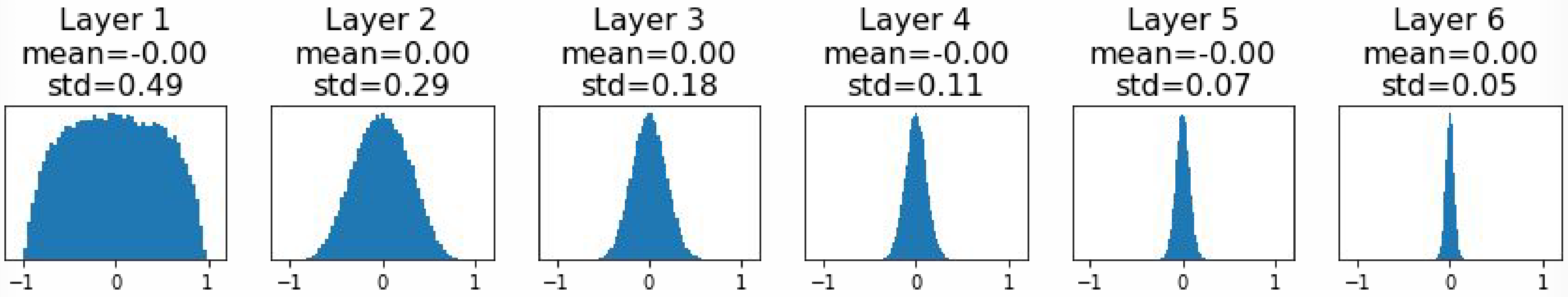
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

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 =(



Slide Credit cs231n

Weight Initialization: Activation statistics

```
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

Weight Initialization: Activation statistics

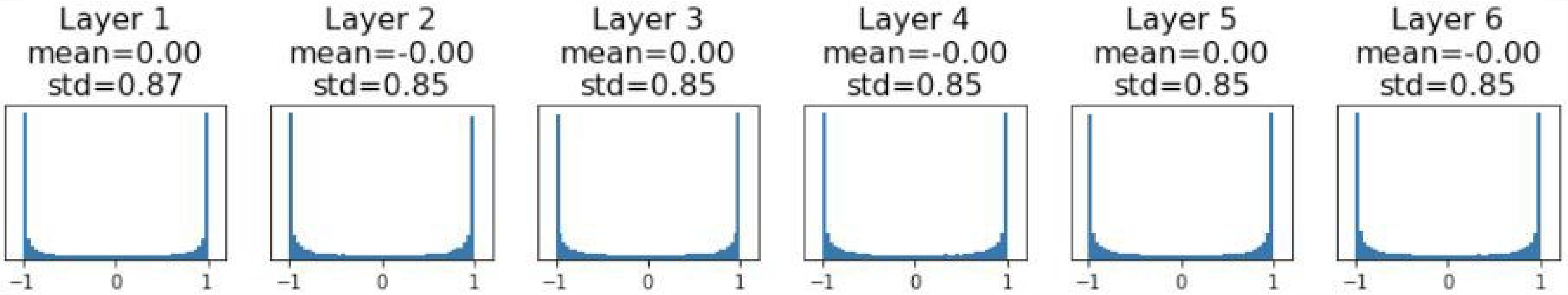
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Increase std of initial weights from 0.01 to 0.05

All activations saturate

Q: What do the gradients look like?

A: Local gradients all zero, no learning =(



Slide Credit cs231n

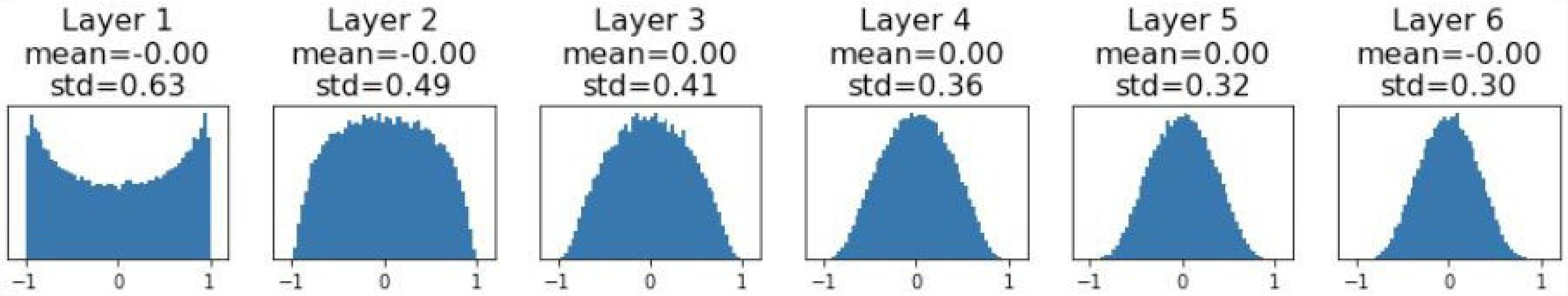
Weight Initialization: "Xavier" Initialization

```
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(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!

For conv layers, Din is $kernel\_size^2$ * input_channels



Glorot and Bengio, "Understanding the difficulty of training deep feedforward neural networks", AISTAT 2010

Slide Credit cs231n

Weight Initialization: "Xavier" 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(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!

For conv layers, Din is $kernel\_size^2$ * input_channels

Derivation:

$y = Wx$
$h = f(y)$

$\mathrm{Var}(y_i) = \mathrm{Din} * \mathrm{Var}(x_i w_i)$ [Assume x, w are iid]

$= \mathrm{Din} * (E[x_i^2]E[w_i^2] - E[x_i]^2 E[w_i]^2)$ [Assume x, w independant]

$= \mathrm{Din} * \mathrm{Var}(x_i) * \mathrm{Var}(w_i)$ [Assume x, w are zero-mean]

If $\mathrm{Var}(w_i) = 1/\mathrm{Din}$ then $\mathrm{Var}(y_i) = \mathrm{Var}(x_i)$

Glorot and Bengio, "Understanding the difficulty of training deep feedforward neural networks", AISTAT 2010

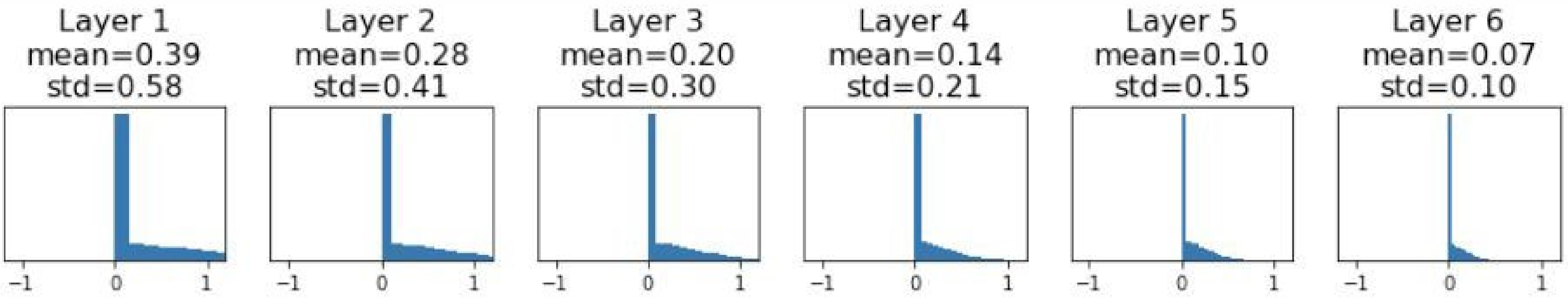
Weight Initialization: What about ReLU?

```
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 =(



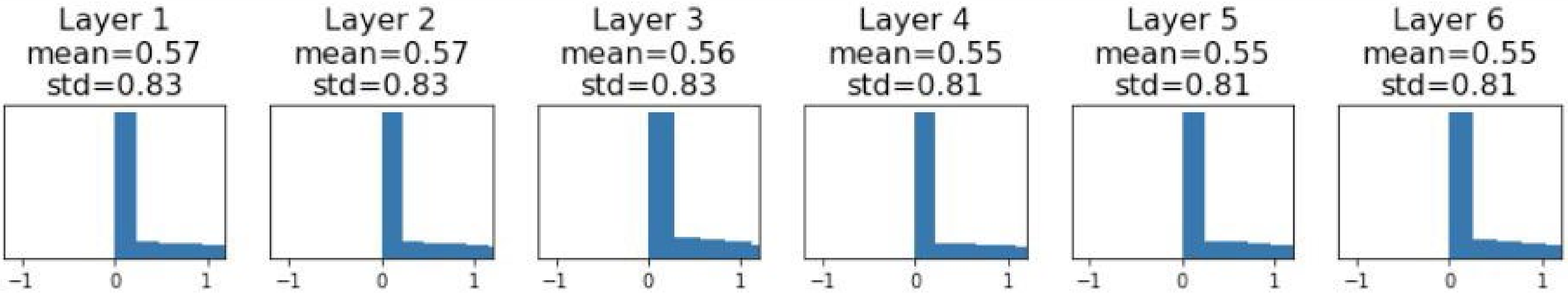
Slide Credit cs231n

Weight Initialization: Kaiming / 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!



He et al, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification", ICCV 2015

Proper initialization is 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 Initia0lizations of Convolutional Neural Networks by Krähenbühl et al., 2015
- All you need is a good init, Mishkin and Matas, 2015
- Fixup Initialization: Residual Learning Without Normalization, Zhang et al, 2019
- The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks, Frankle and Carbin, 2019
- Hendrycks, Dan, and Kevin Gimpel. "Gaussian error linear units (gelus)." arXiv preprint arXiv:1606.08415 (2016).

BATCH NORMALIZATION

Batch Normalization

"you want zero-mean unit-variance activations? just make them so."

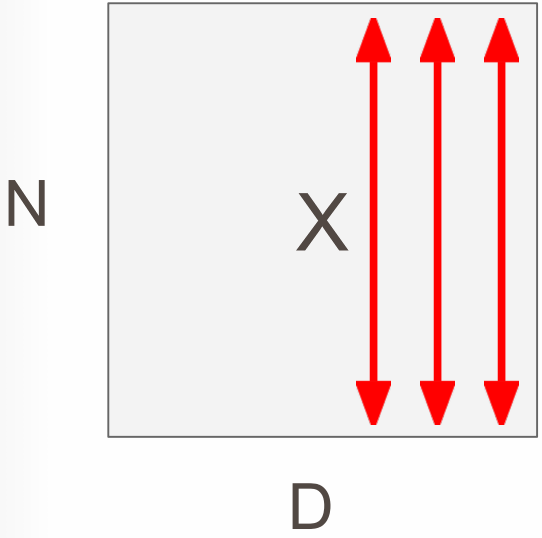
consider a batch of activations at some layer. To make each dimension zero-mean unit-variance, apply:

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

[Ioffe and Szegedy, 2015]

Batch Normalization

Input: $x : N \times D$



[Ioffe and Szegedy, 2015]

$$\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}$$

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

Problem: What if zero-mean, unit variance is too hard of a constraint?

Batch Normalization

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

Input: $x : N \times D$

Learnable scale and shift parameters:

$\gamma, \beta : D$

$\mu_j =$ (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

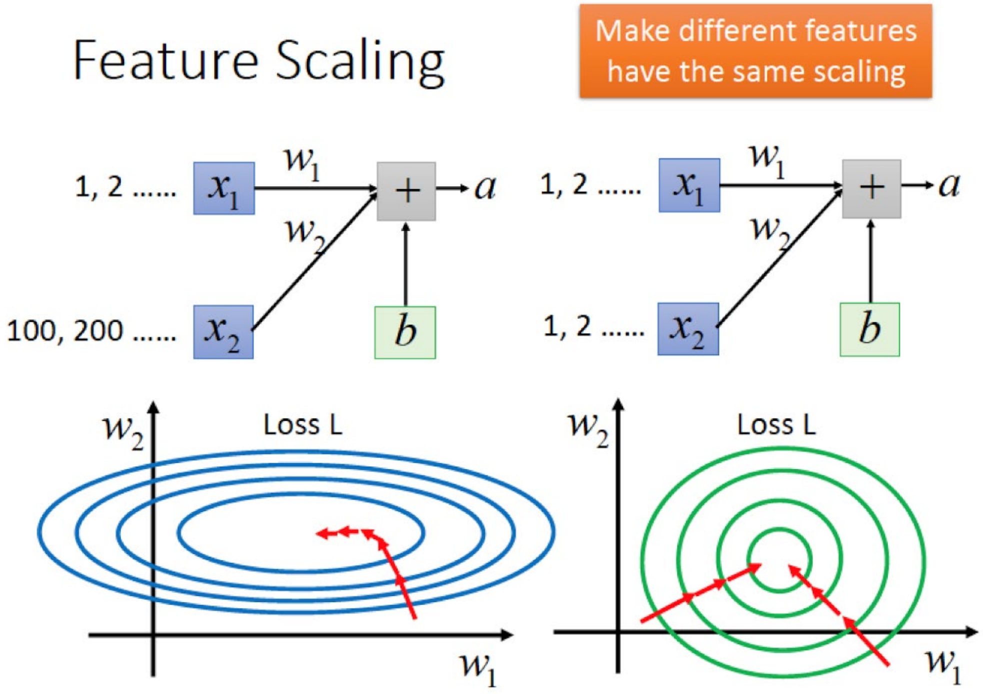
$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

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

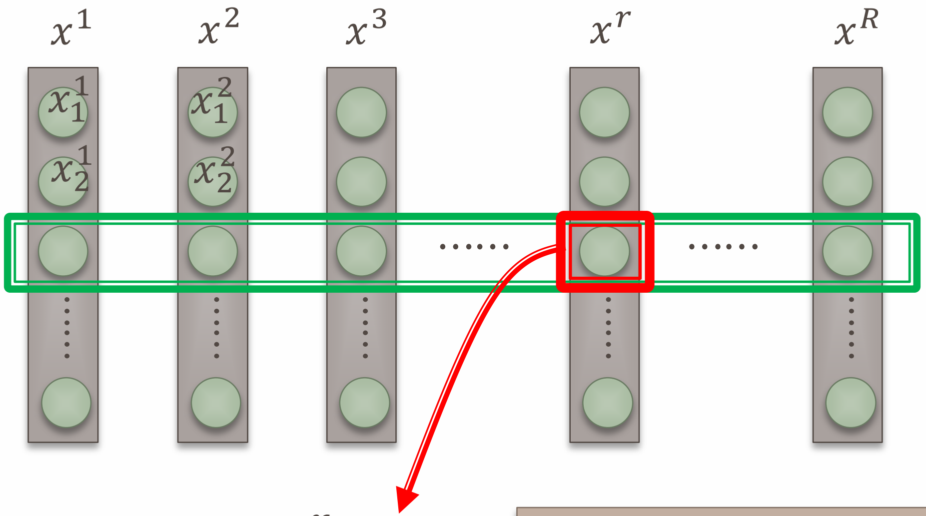
[Ioffe and Szegedy, 2015]

Batch Normalization



Slide Credit cs231n

Feature Scaling



For each dimension i:

mean: m_i

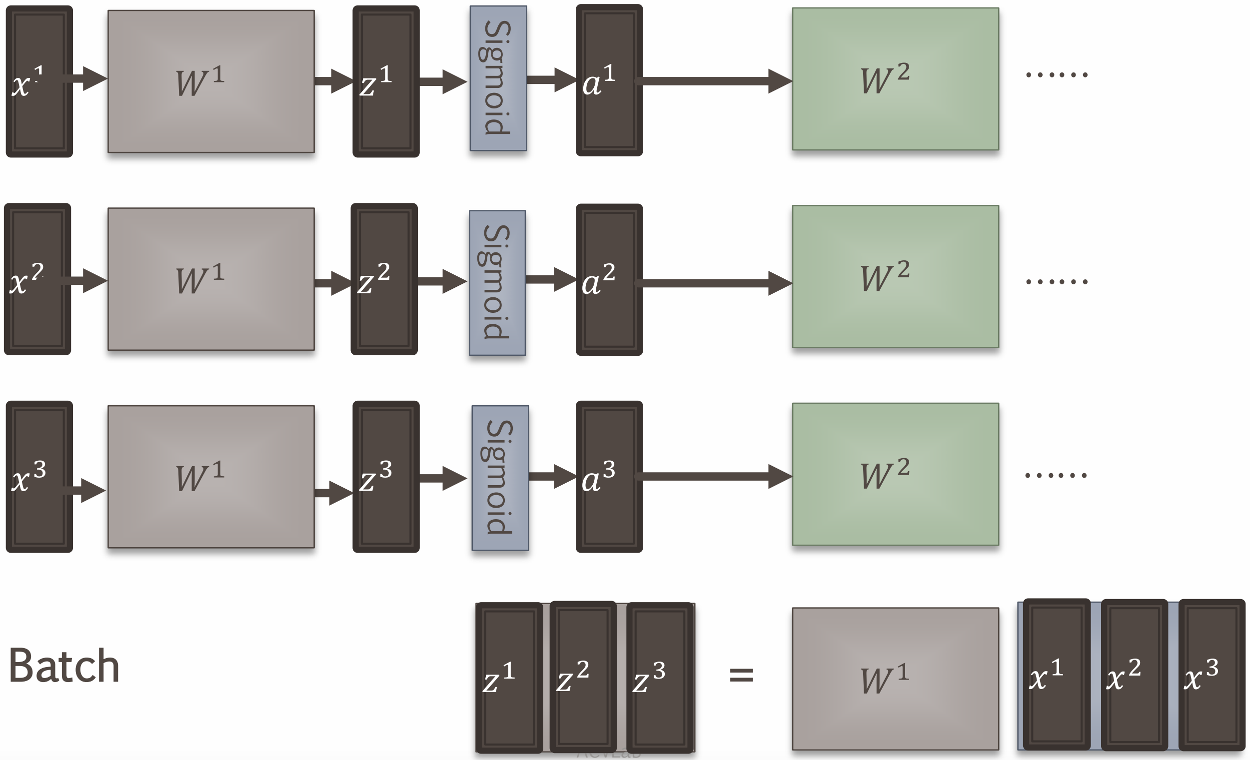
standard deviation: σ_i

$$x_i^r \leftarrow \frac{x_i^r - m_i}{\sigma_i}$$

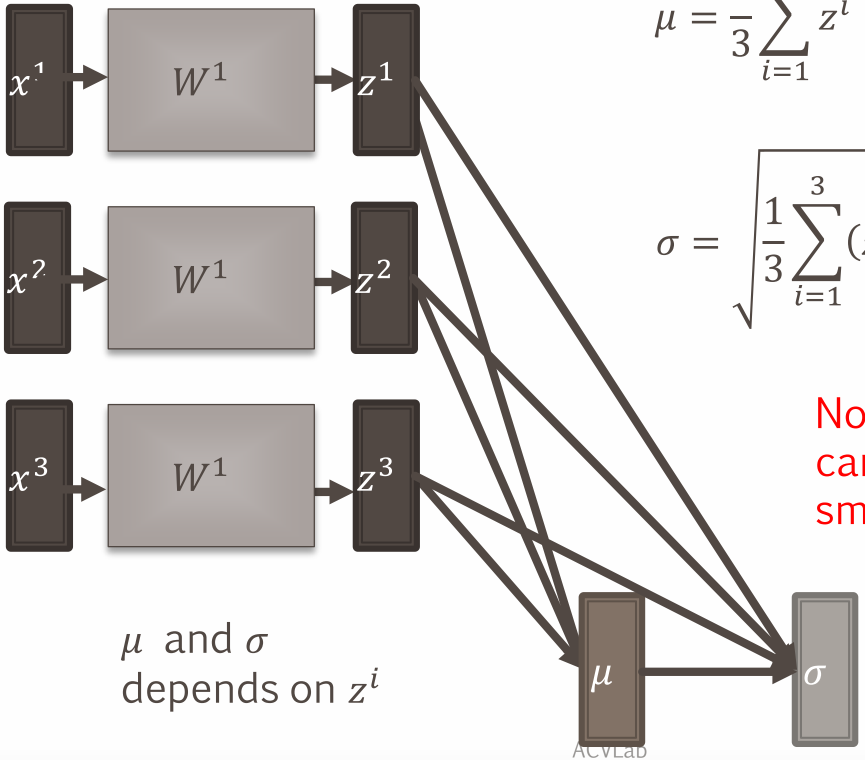
The means of all dimensions are 0, and the variances are all 1

In general, gradient descent converges much faster with feature scaling than without it.

Batch Normalization



Batch normalization



$$\mu = \frac{1}{3}\sum_{i=1}^{3} z^i$$

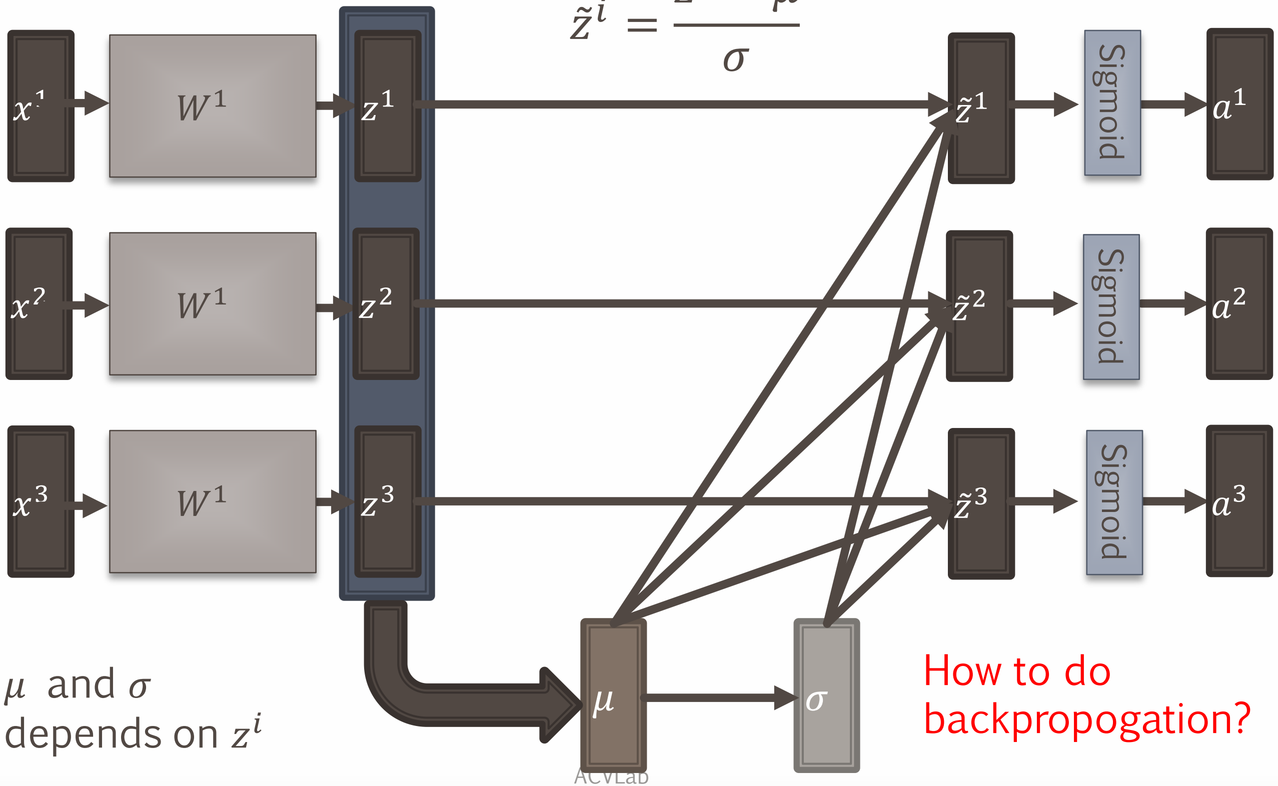
$$\sigma = \sqrt{\frac{1}{3}\sum_{i=1}^{3}(z^i - \mu)^2}$$

Note: Batch normalization cannot be applied on small batch.

μ and σ depends on z^i

Batch normalization

$$\tilde{z}^i = \frac{z^i - \mu}{\sigma}$$

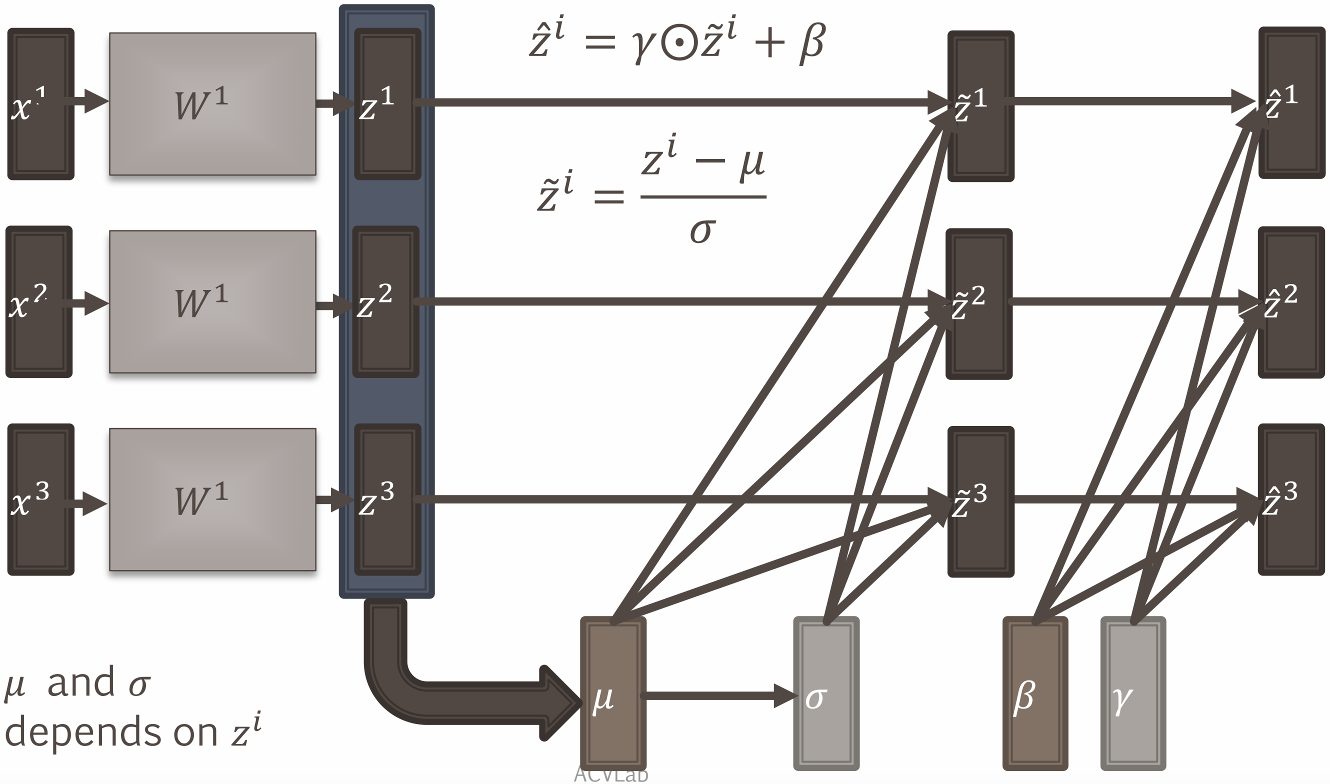


μ and σ depends on z^i

How to do backpropogation?

Slide Credit cs231n

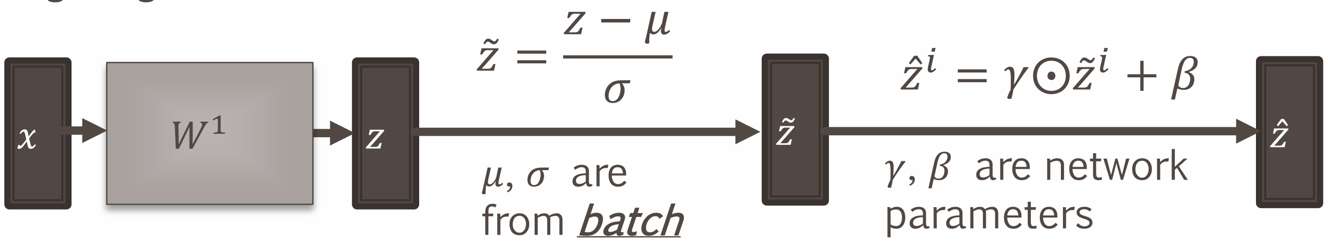
Batch normalization



μ and σ depends on z^i

Batch normalization

- At testing stage:



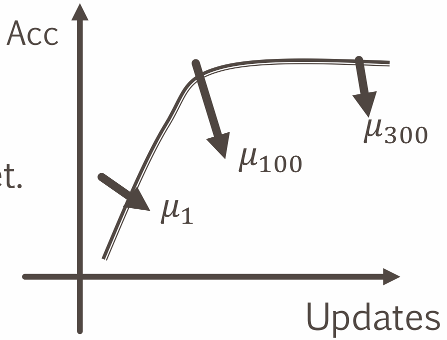
We do not have ***batch*** at testing stage.

Ideal solution:

Computing μ and σ using the whole training dataset.

Practical solution:

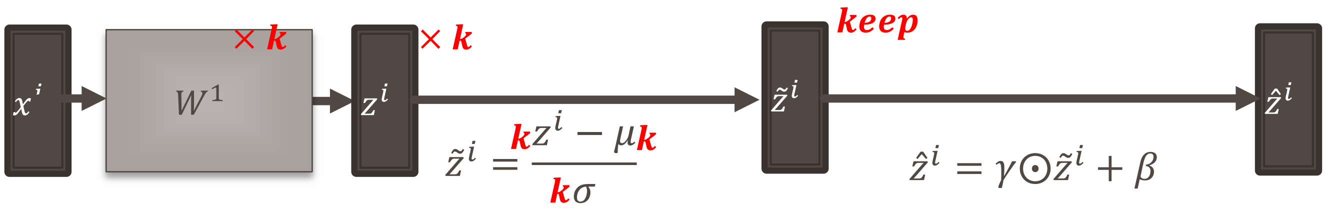
Computing the moving average of μ and σ of the batches during training.



Slide Credit cs231n

Batch normalization - Benefit

- BN reduces training times, and make very deep net trainable.
  - Because of less Covariate Shift, we can use larger learning rates.
  - Less exploding/vanishing gradients
    - Especially effective for sigmoid, tanh, etc.
- Learning is less affected by initialization.



- BN reduces the demand for regularization.

Batch Normalization for ConvNets

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)

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

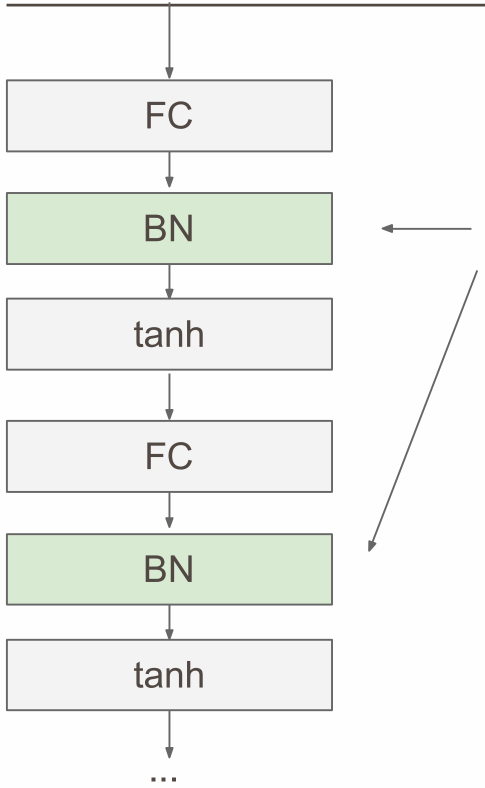
Normalize

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

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

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

Batch Normalization

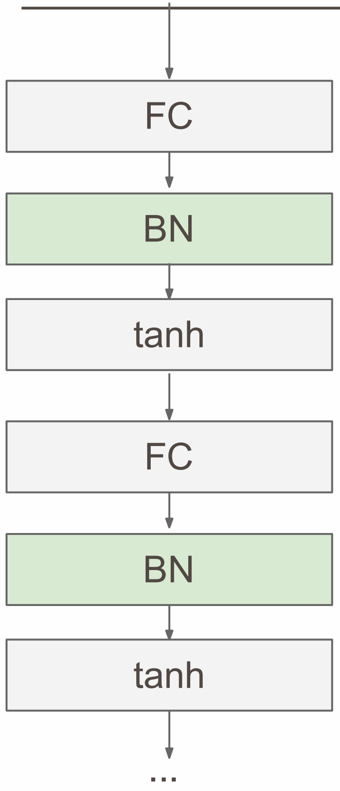


Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

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

[Ioffe and Szegedy, 2015]

Batch Normalization



- 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 is a very common source of bugs!

Layer Normalization

Batch Normalization for fully-connected networks

$\mathbf{x}: N \times D$

$$\boxed{\begin{array}{l}\text{Normalize} \\ \boldsymbol{\mu}, \boldsymbol{\sigma}: 1 \times D\end{array}}$$

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

$\mathbf{y} = \gamma(\mathbf{x}-\boldsymbol{\mu})/\boldsymbol{\sigma}+\beta$

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

$\mathbf{x}: N \times D$

$$\boxed{\begin{array}{l}\text{Normalize} \\ \boldsymbol{\mu}, \boldsymbol{\sigma}: N \times 1\end{array}}$$

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

$\mathbf{y} = \gamma(\mathbf{x}-\boldsymbol{\mu})/\boldsymbol{\sigma}+\beta$

Ba, Kiros, and Hinton, "Layer Normalization", arXiv 2016

Instance Normalization

Batch Normalization for convolutional networks

$$\mathbf{x}: \ N \times C \times H \times W$$

Normalize

$$\boldsymbol{\mu}, \boldsymbol{\sigma}: \ 1 \times C \times 1 \times 1$$

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

$$\mathbf{y} = \gamma(\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \beta$$

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

$$\mathbf{x}: \ N \times C \times H \times W$$

Normalize

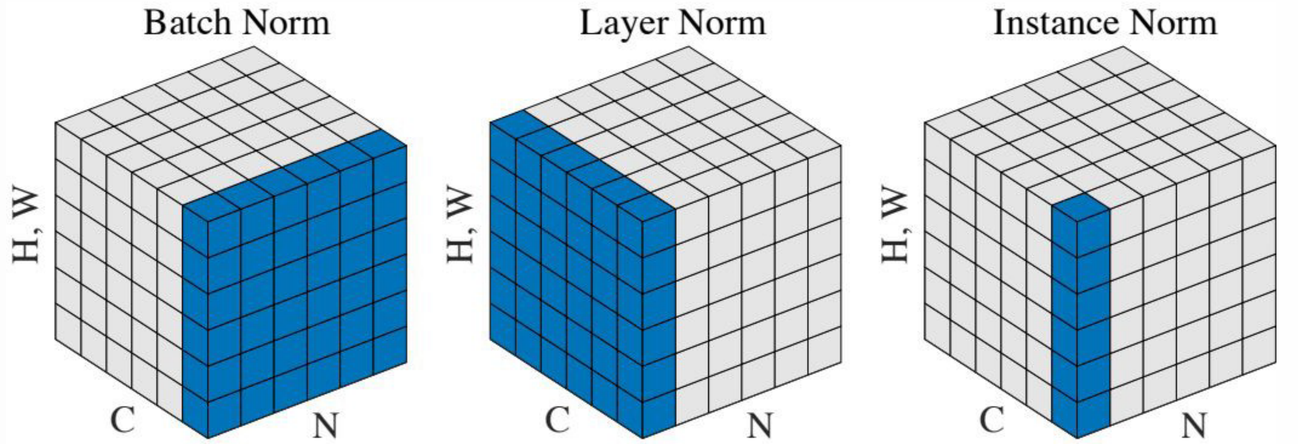
$$\boldsymbol{\mu}, \boldsymbol{\sigma}: \ N \times C \times 1 \times 1$$

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

$$\mathbf{y} = \gamma(\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \beta$$

Ulyanov et al, Improved Texture Networks: Maximizing Quality and Diversity in Feed-forward Stylization and Texture Synthesis, CVPR 2017

Comparison of Normalization Layers



$\mathcal{S}_i = \{k \mid k_C = i_C\},$

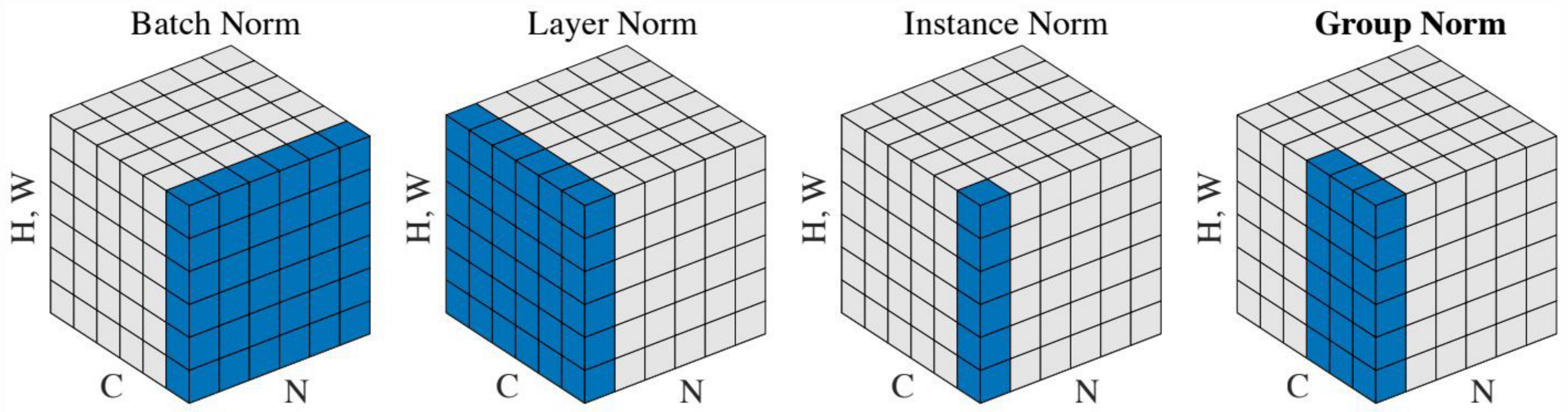
$\mathcal{S}_i = \{k \mid k_N = i_N\},$

$\mathcal{S}_i = \{k \mid k_N = i_N, k_C = i_C\}.$

Wu and He, “Group Normalization”, ECCV 2018

Slide Credit cs231n

Group Normalization



$\mathcal{S}_i = \{k \mid k_C = i_C\}$, $\quad \mathcal{S}_i = \{k \mid k_N = i_N\}$, $\quad \mathcal{S}_i = \{k \mid k_N = i_N, k_C = i_C\}$. $\quad \{k \mid k_N = i_N, \lfloor \frac{k_C}{C/G} \rfloor = \lfloor \frac{i_C}{C/G} \rfloor\}$.

Wu and He, “Group Normalization”, ECCV 2018

Summary

- We looked in detail at:
- Activation Functions (use ReLU)
- Data Preprocessing (images: subtract mean)
- Weight Initialization (use Xavier/He init)
- Batch Normalization (use)
- Advanced:
  - Spectral normalization! Avoid the gradient vary significantly!

Algorithm 1 SGD with spectral normalization

- Initialize $\tilde{\boldsymbol{u}}_l \in \mathcal{R}^{d_l}$ for $l = 1, \ldots, L$ with a random vector (sampled from isotropic distribution).
- For each update and each layer l:
  1. Apply power iteration method to a unnormalized weight W^l:

$$\tilde{\boldsymbol{v}}_l \leftarrow (W^l)^{\mathrm{T}} \tilde{\boldsymbol{u}}_l / \|(W^l)^{\mathrm{T}} \tilde{\boldsymbol{u}}_l\|_2 \quad (20)$$

$$\tilde{\boldsymbol{u}}_l \leftarrow W^l \tilde{\boldsymbol{v}}_l / \|W^l \tilde{\boldsymbol{v}}_l\|_2 \quad (21)$$

  2. Calculate $\bar{W}_{\mathrm{SN}}$ with the spectral norm:

$$\bar{W}^l_{\mathrm{SN}}(W^l) = W^l / \sigma(W^l), \text{ where } \sigma(W^l) = \tilde{\boldsymbol{u}}_l^{\mathrm{T}} W^l \tilde{\boldsymbol{v}}_l \quad (22)$$

  3. Update W^l with SGD on mini-batch dataset $\mathcal{D}_M$ with a learning rate α:

$$W^l \leftarrow W^l - \alpha \nabla_{W^l} \ell(\bar{W}^l_{\mathrm{SN}}(W^l), \mathcal{D}_M) \quad (23)$$

Next: How to train NN effectively and efficiently?

- Parameter update schemes
- Learning rate schedules
- Gradient checking
- Regularization (Dropout etc.)
- Learning scheduler
- Hyperparameter setting/search
- Evaluation (Ensembles etc.)
- Transfer learning / fine-tuning