Neural Networks 301

Deep Learning for Natural Language Processing

Vladislav Lialin, Text Machine Lab

Administrative

- Python Code style quiz is due now
- Homework 3 is due Thursday before the class

What you should remember after this lecture

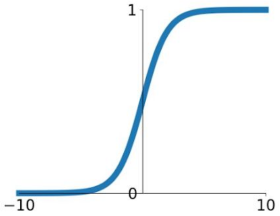
- Don’t worry about activations too much
  - But don’t use sigmoid unless you have reasons to
  - ReLU is the default choice
- Weight initialization matters
- ADAM optimizer
- Batch Normalization
- Dropout
- Neural networks are hard to train

Activation functions

Activation functions

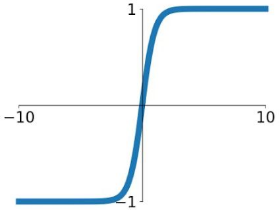
Sigmoid

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



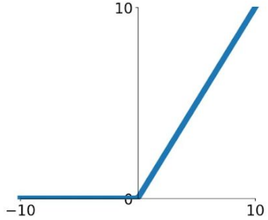
tanh

$\tanh(x)$



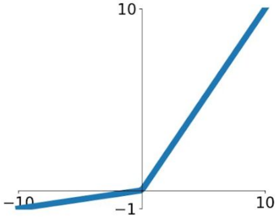
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

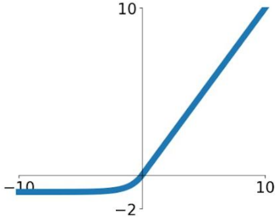


Maxout

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

ELU

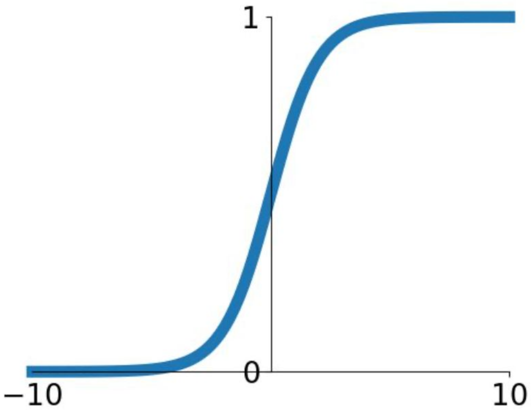
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Slide credit: Stanford CS231n
Image credit: Stanford CS231n

Sigmoid

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



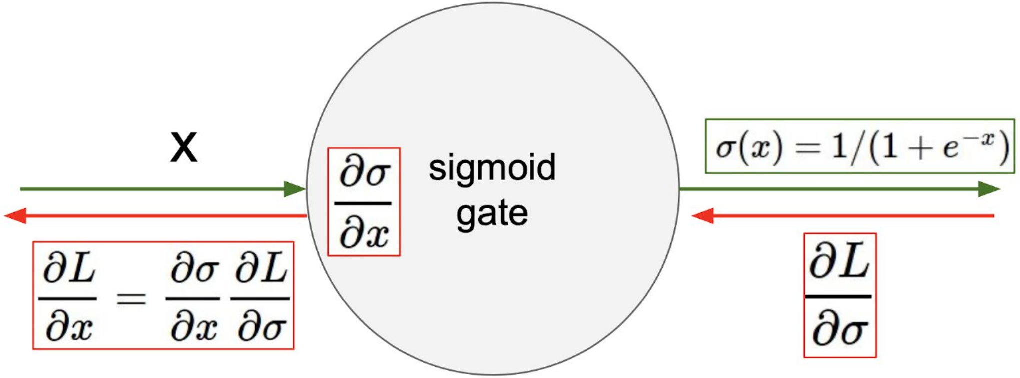
- Squashes numbers to range [0, 1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

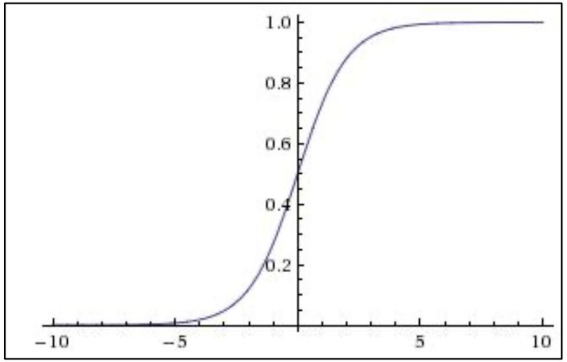
3 problems

- Saturated neurons “kill” gradients, which does not allow to update network parameters

Slide credit: Stanford CS231n
Image credit: Stanford CS231n

Sigmoid



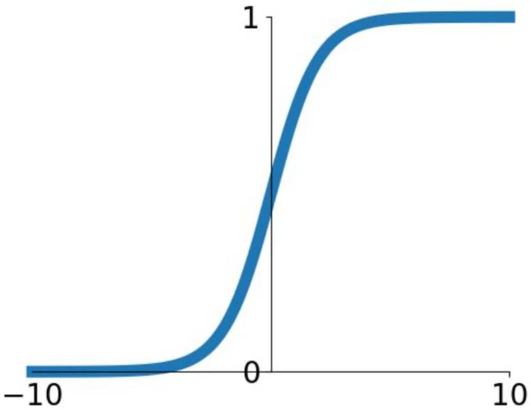


What happens when x = -10?
What happens when x = 0?
What happens when x = 10?

Slide credit: Stanford CS231n
Image credit: Stanford CS231n

Sigmoid

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



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

3 problems

- Saturated neurons "kill" gradients, which does not allow to update network parameters
- Not zero-centered — training converges slower

Slide credit: Stanford CS231n
Image credit: Stanford CS231n

Illustration why zero-centricity matters

Not a proof, but hand-waving some of the intuition

- Imagine you have a fully-connected network with sigmoid layers
- The output of the first layer will only have positive numbers
- Let's look at a single neuron of the second layer
  - What happens with the gradient for the weights w?

Illustration why zero-centricity matters

Not a proof, but hand-waving some of the intuition

- Imagine you have a fully-connected network with sigmoid layers
- The output of the first layer will only have positive numbers
- Let's look at a single neuron of the second layer
    - What happens with the gradient for the weights w?

$$\frac{dL}{dw} = \sigma(\sum_i x_i w_i + b)(1 - \sigma(\sum_i x_i w_i + b)) \cdot x \cdot upstream\_grad$$

Slide credit: Stanford CS231n

Illustration why zero-centricity matters

Not a proof, but hand-waving some of the intuition

- Imagine you have a fully-connected network with sigmoid layers
- The output of the first layer will only have positive numbers
- Let's look at a single neuron of the second layer
  - What happens with the gradient for the weights w?

Always positive

$$\frac{dL}{dw} = \boxed{\sigma(\sum_i x_i w_i + b)(1 - \sigma(\sum_i x_i w_i + b))} \cdot x \cdot upstream\_grad$$

Slide credit: Stanford CS231n

Illustration why zero-centricity matters

Not a proof, but hand-waving some of the intuition

- Imagine you have a fully-connected network with sigmoid layers
- The output of the first layer will only have positive numbers
- Let's look at a single neuron of the second layer
  - What happens with the gradient for the weights w?

Always positive

Positive because x is the output of the previous sigmoid layer

$$\frac{dL}{dw} = \underbrace{\sigma(\sum_i x_i w_i + b)(1 - \sigma(\sum_i x_i w_i + b))}_{\text{Always positive}} \cdot \underbrace{x}_{\text{Positive}} \cdot upstream\_grad$$

Slide credit: Stanford CS231n

Illustration why zero-centricity matters

Not a proof, but hand-waving some of the intuition

- Imagine you have a fully-connected network with sigmoid layers
- The output of the first layer will only have positive numbers
- Let's look at a single neuron of the second layer
    - What happens with the gradient for the weights w?

Always positive

Positive because x is the output of the previous sigmoid layer

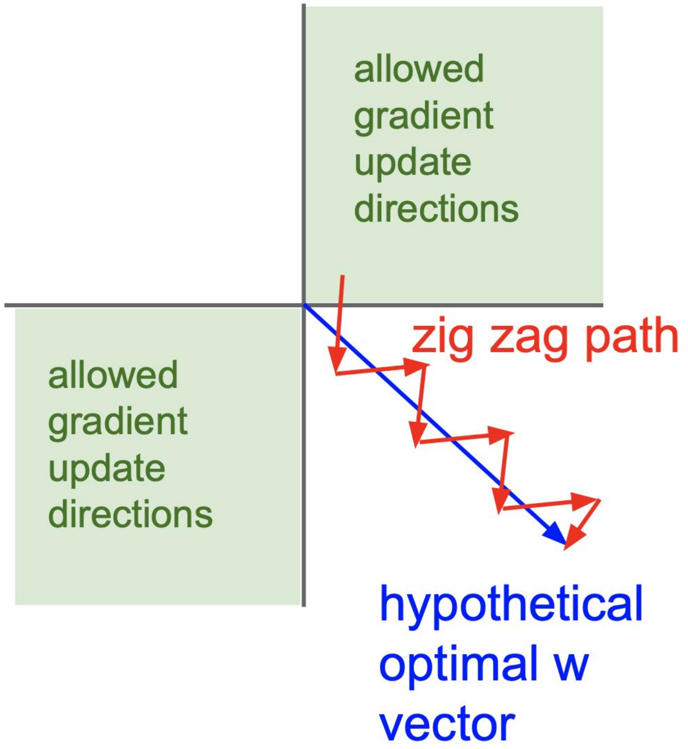
$$\boxed{\frac{dL}{dw}} = \boxed{\sigma(\sum_i x_i w_i + b)(1 - \sigma(\sum_i x_i w_i + b))} \cdot \boxed{x} \cdot \boxed{upstream\_grad}$$

Same sign for all of the w of this layer

Slide credit: Stanford CS231n

Illustration why zero-centricity matters

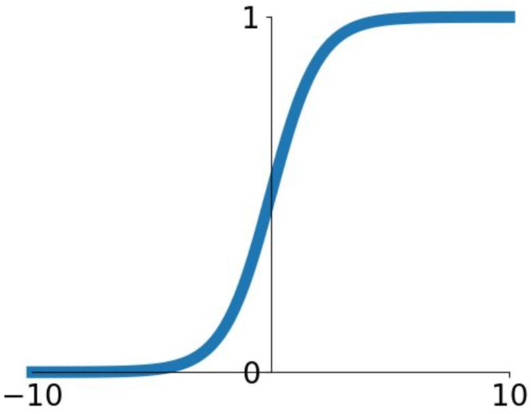
- Gradients for all w for a given neuron will have the same sign
- However, if you consider computing gradients over multiple examples (as usually done on practice), it is not a big problem



Slide credit: Stanford CS231n
Image credit: Stanford CS231n

Sigmoid

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



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

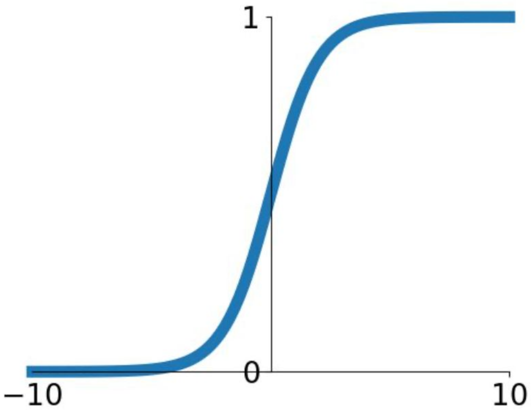
3 problems

- Saturated neurons “kill” gradients, which does not allow to update network parameters
- Not zero-centered — training converges slower
- exp() is a bit expensive to compute

Slide credit: Stanford CS231n
Image credit: Stanford CS231n

Sigmoid

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



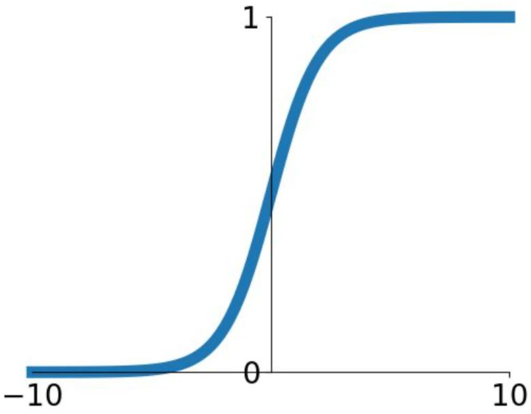
- Squashes numbers to range [0, 1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

3 problems

- **Saturated neurons "kill" gradients, which does not allow to update network parameters**
- Not zero-centered — training converges slower
- exp() is a bit expensive to compute

Slide credit: Stanford CS231n
Image credit: Stanford CS231n

Tanh



- Just a sigmoid times 2 minus one
- Squashes numbers to range [-1, 1]
- Zero-centered

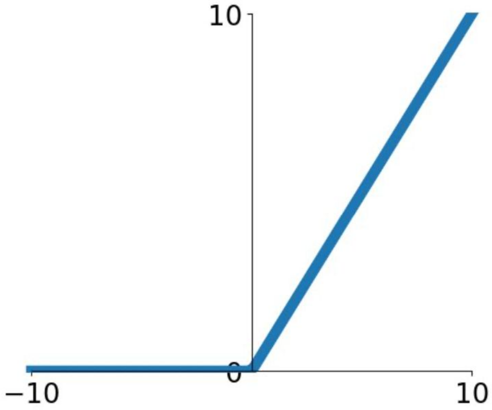
2 problems

- Saturated neurons “kill” gradients, which does not allow to update network parameters
- exp() is a bit expensive to compute

Slide credit: Stanford CS231n
Image credit: Stanford CS231n

Rectified Linear Unit (ReLU)

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



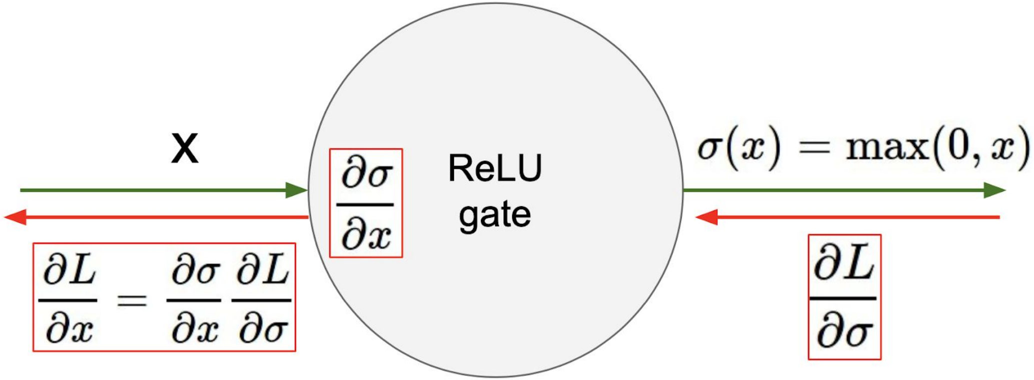
- Does not saturate
    - Converges much faster than sigmoid tanh on practice
- Very computationally efficient

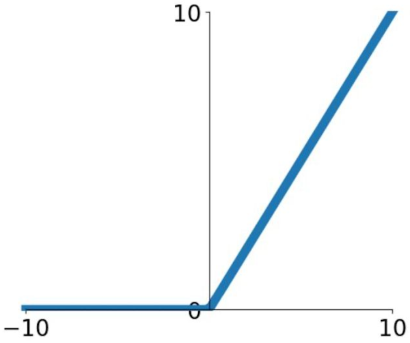
2 problems

- Non zero-centered output
- “Dead neurons”

Slide credit: Stanford CS231n
Image credit: Stanford CS231n

ReLU: Dead Neurons

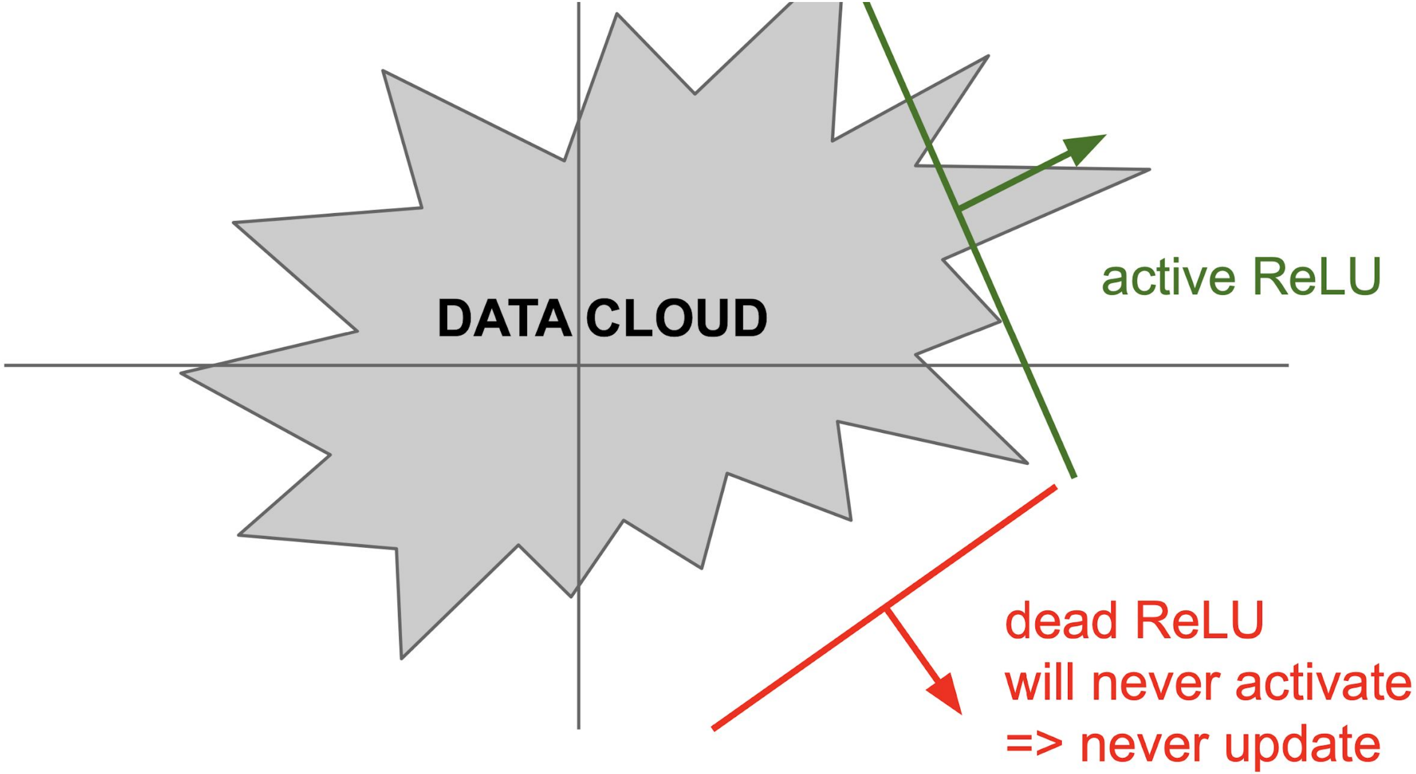




What happens when x = -10?
What happens when x = 0?
What happens when x = 10?

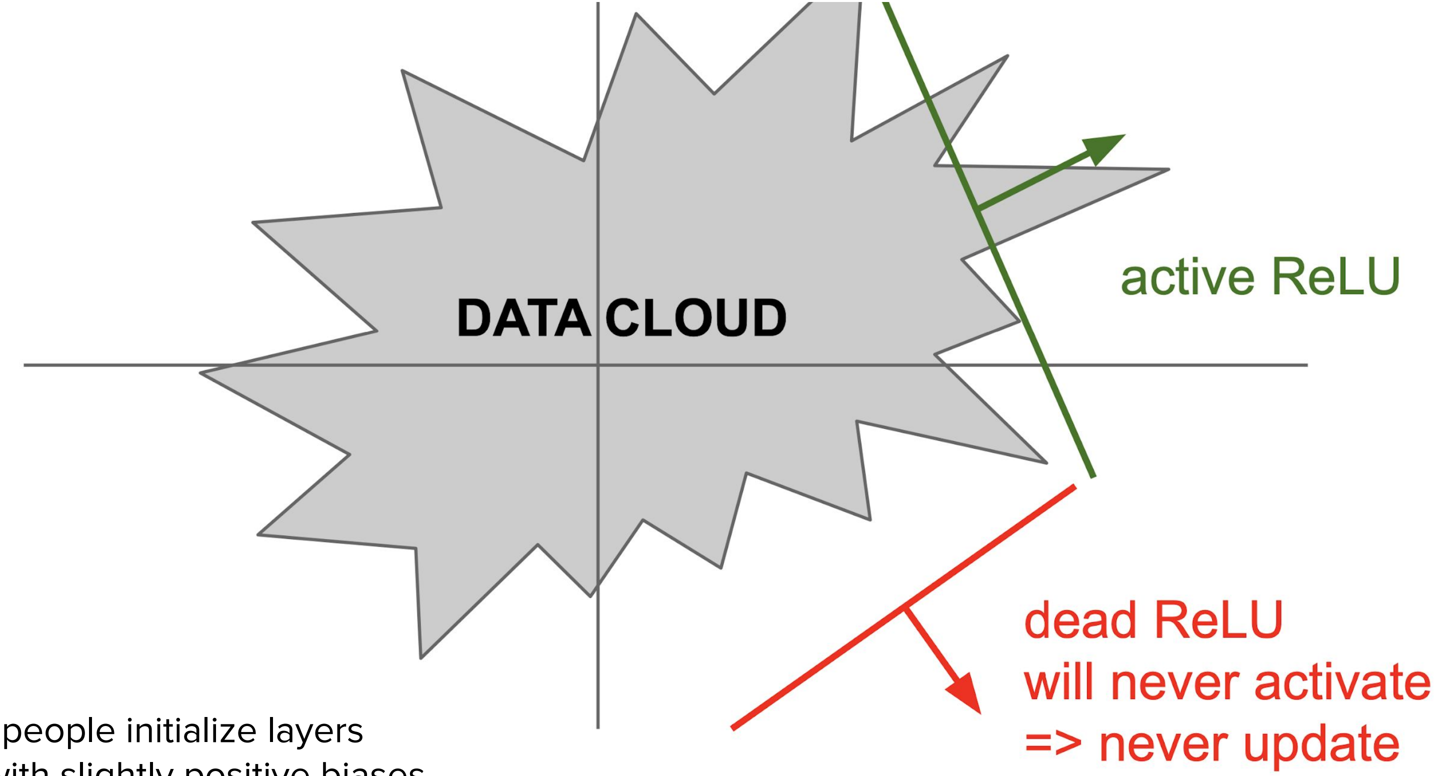
Slide credit: Stanford CS231n
Image credit: Stanford CS231n

ReLU: Dead Neurons



Slide credit: Stanford CS231n
Image credit: Stanford CS231n

ReLU: Dead Neurons

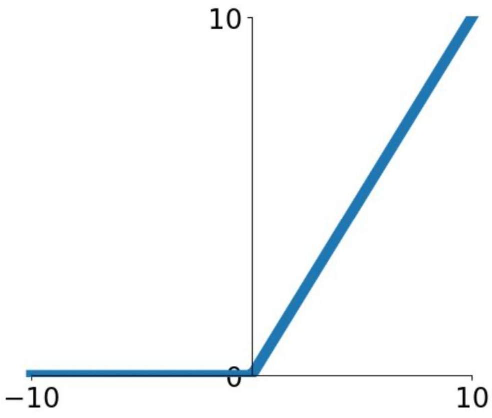


Sometimes people initialize layers with ReLU with slightly positive biases to mitigate this.

Slide credit: Stanford CS231n
Image credit: Stanford CS231n

Leaky ReLU

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



- Does not saturate
  - Converges much faster than sigmoid tanh in practice
- Very computationally efficient
- Should not kill neurons

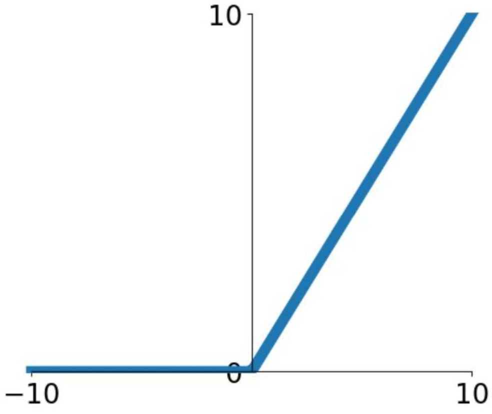
2 problems

- Non zero-centered output
- Performs on-par with ReLU in practice 🤷‍♂️

Slide credit: Stanford CS231n
Image credit: Stanford CS231n

Parametric Leaky ReLU

$$LeakyReLU(\alpha, x) = \max(\alpha x,\ x),\ \ \alpha \in \mathbb{R}$$



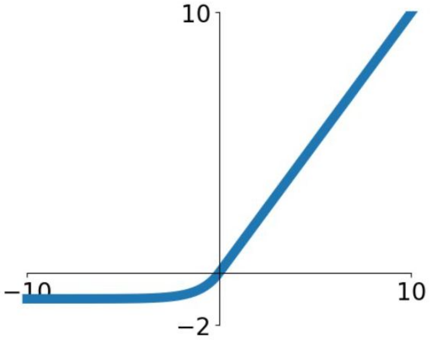
- Does not saturate
  - Converges much faster than sigmoid tanh in practice
- Very computationally efficient
- Should not kill neurons
- When tuned for alpha, performs better than ReLU

2 problems

- Non zero-centered output
- Requires expensive hparam tuning for alpha

Slide credit: Stanford CS231n
Image credit: Stanford CS231n

Exponential Linear Unit (ELU)



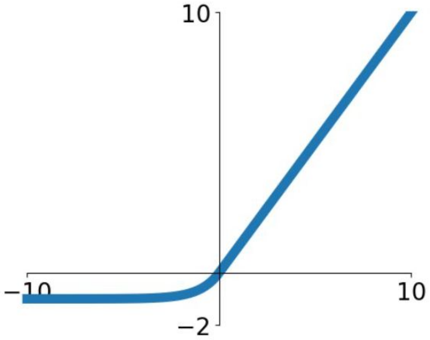
- All benefits of ReLU
- Closer to zero mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise

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

- Computation requires exp()

Slide credit: Stanford CS231n
Image credit: Stanford CS231n

Exponential Linear Unit (ELU)



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

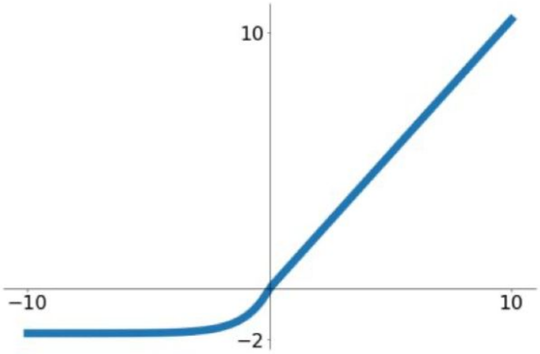
- All benefits of ReLU
- Closer to zero mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise

Not a problem anymore if you use CUDA cores

- ~~Computation requires exp()~~

Slide credit: Stanford CS231n
Image credit: Stanford CS231n

Scaled Exponential Linear Unit (SELU)



$$f(x) = \begin{cases} \lambda x & \text{if } x > 0 \\ \lambda\alpha(e^x - 1) & \text{otherwise} \end{cases}$$

$\alpha = 1.6733$, $\lambda = 1.0507$

- Scaled versionof ELU that works better for deep networks
- "Self-normalizing" property;
- Can train deep SELU networks without BatchNorm
    - (will discuss more later)

Slide credit: Stanford CS231n
Image credit: Stanford CS231n

Gaussian Error Linear Unit

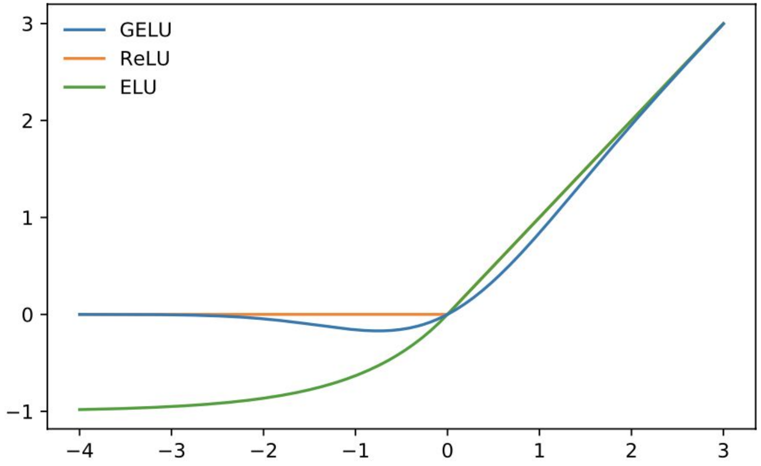


Figure 1: The GELU ($\mu = 0, \sigma = 1$), ReLU, and ELU ($\alpha = 1$).

$$GELU(x) = x \cdot \frac{1}{2}\left[1 + erf(\frac{x}{\sqrt{2}})\right] \approx x\sigma(1.702x)$$

- Usually works better than ReLU for large networks (>10 layers)
- Non-monotoric — both negative and positive values of the gradient

2 problems

- Non zero-centered output
- Expensive to compute

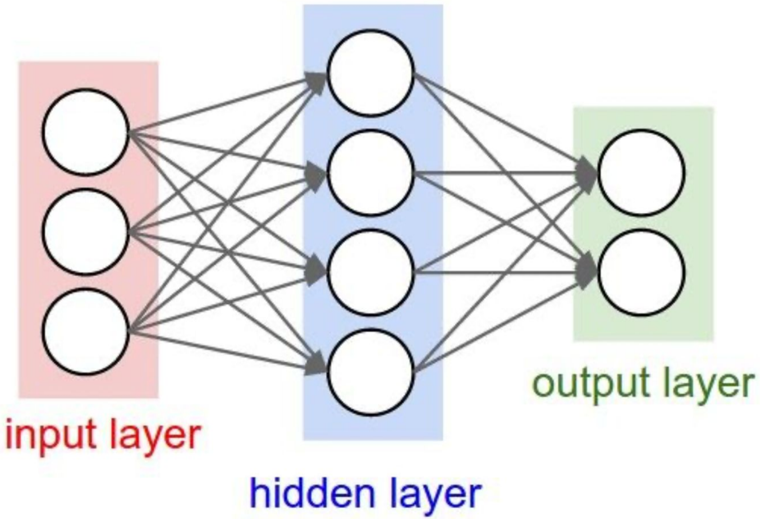
TL;DR

- ReLU is a good default choice
- Explore GELU, Parametric Leaky ReLU, SeLU, Swish, ...
- Do not use sigmoid/tanh unless you want probabilities

Activation functions is an active area of research. You can experiment with implementing your own activation function.

Weight initialization

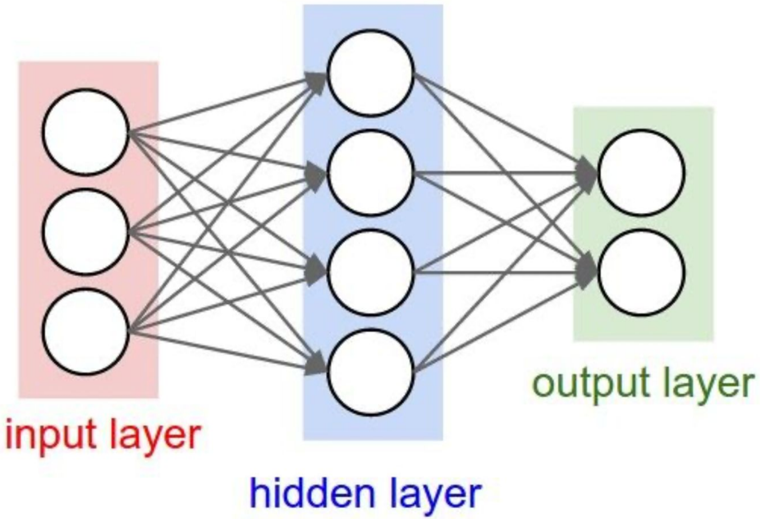
- Q: what happens when W=constant init is used?



Slide credit: Stanford CS231n
Image credit: Stanford CS231n

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

$$y_1 = xW_1 + b_1$$
$$y_2 = y_1 W_2 + b_2$$
$$l = l(y_2, y_{target})$$
$$\frac{dl}{dW_2^{i,j}} = \frac{dl}{dy_2^j} \cdot \frac{dy_2^j}{dW_2^{i,j}}$$



If all Wij are the same, then gradients will be **different** for Wij across **i**-dimension,
but **exactly the same** across **j**-dimension — hidden dimension.
So effectively hidden dimension is 1 and every **hidden neuron acts exactly the same**.

Slide credit: Stanford CS231n
Image credit: Stanford CS231n

First idea: Small random numbers

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

Works well for small networks, requires tuning the range for large networks

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

What happens to the activations of the first layer?

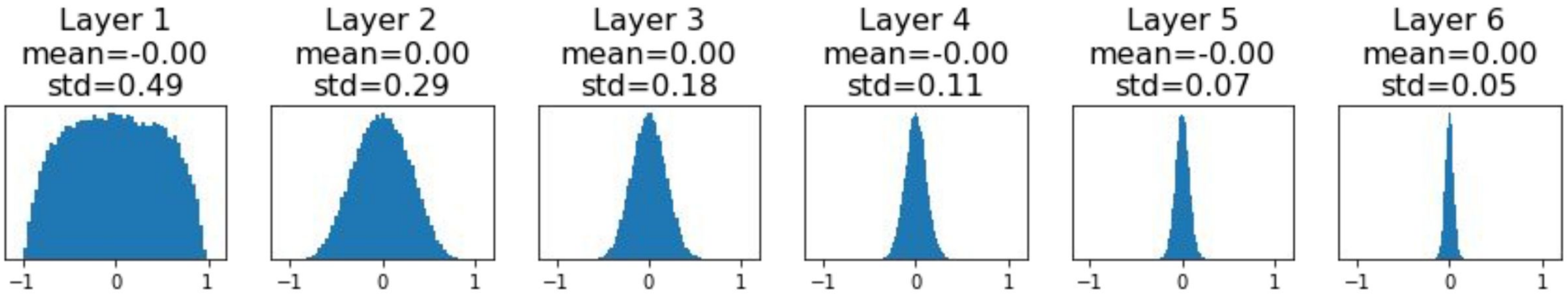
Activation statistics

```
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?



Slide credit: Stanford CS231n
Image credit: Stanford CS231n

Activation statistics

Vanishing gradients

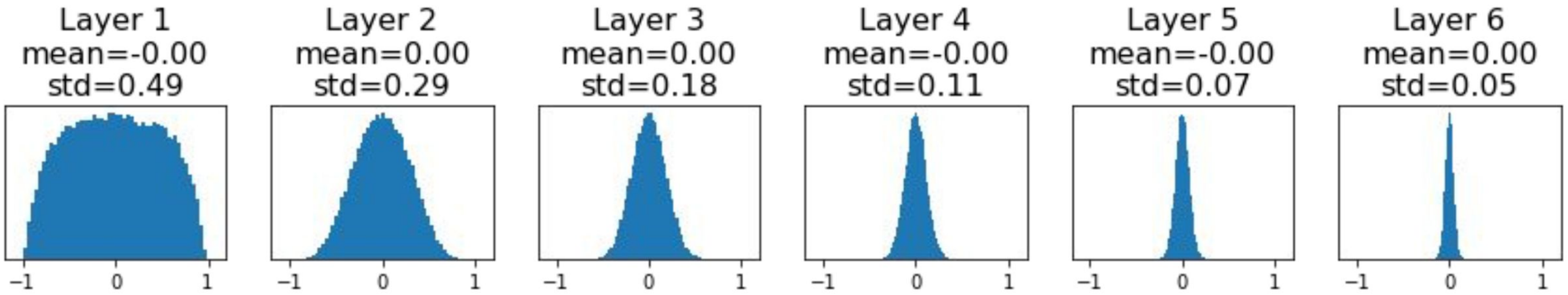
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

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: Zero gradients, no learning



Slide credit: Stanford CS231n
Image credit: Stanford CS231n

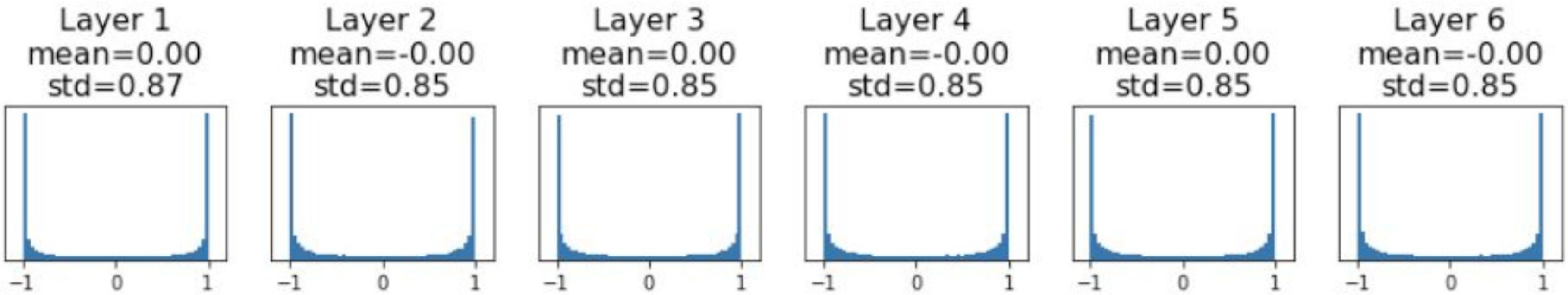
Activation statistics

```
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?



Slide credit: Stanford CS231n
Image credit: Stanford CS231n

Activation statistics

Vanishing gradients

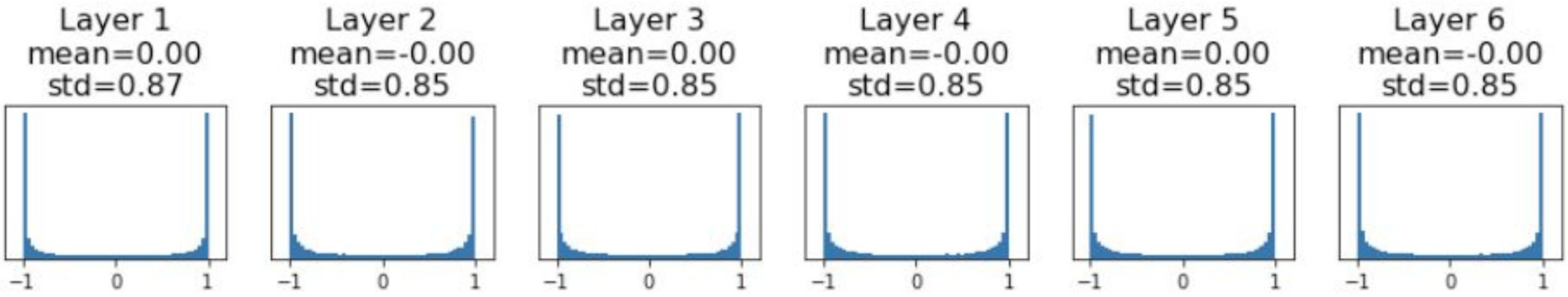
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Increase std of initial weights from 0.01 to 0.05

All activations saturate

Q: What do the gradients look like?

A: Zero local gradients, no learning



Slide credit: Stanford CS231n
Image credit: Stanford CS231n

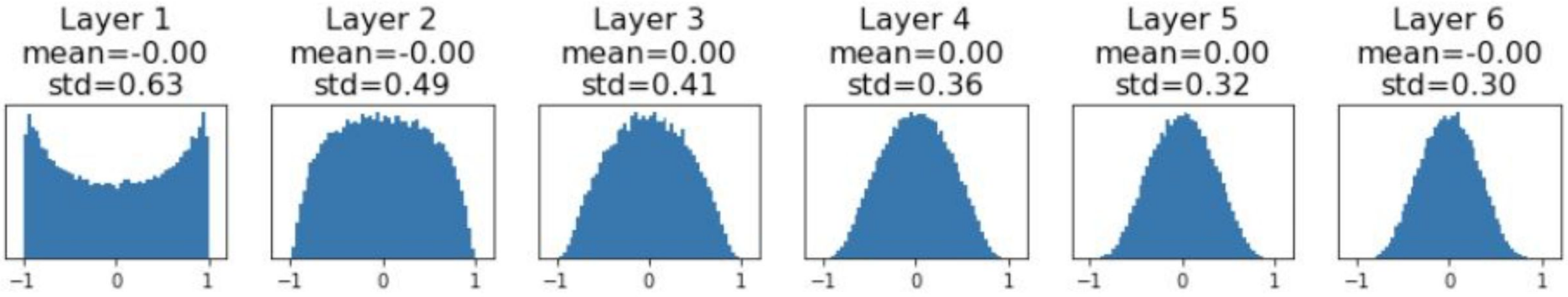
We want the initialization range to be not too big and not too small

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!



Slide credit: Stanford CS231n
Image credit: Stanford CS231n
Understanding the difficulty of training deep feedforward neural networks, Glorot and Bengio, 2010

Xavier Initialization

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

Derivation:

$$
\begin{aligned}
Var(y_i) &= Din * Var(x_i w_i) && \text{[Assume x, w are iid]} \\
&= Din * (E[x_i^2]E[w_i^2] - E[x_i]^2\, E[w_i]^2) && \text{[Assume x, w independant]} \\
&= Din * Var(x_i) * Var(w_i) && \text{[Assume x, w are zero-mean]}
\end{aligned}
$$

If $Var(w_i) = 1/Din$ then $Var(y_i) = Var(x_i)$

Slide credit: Stanford CS231n
Image credit: Stanford CS231n
Understanding the difficulty of training deep feedforward neural networks, Glorot and Bengio, 2010

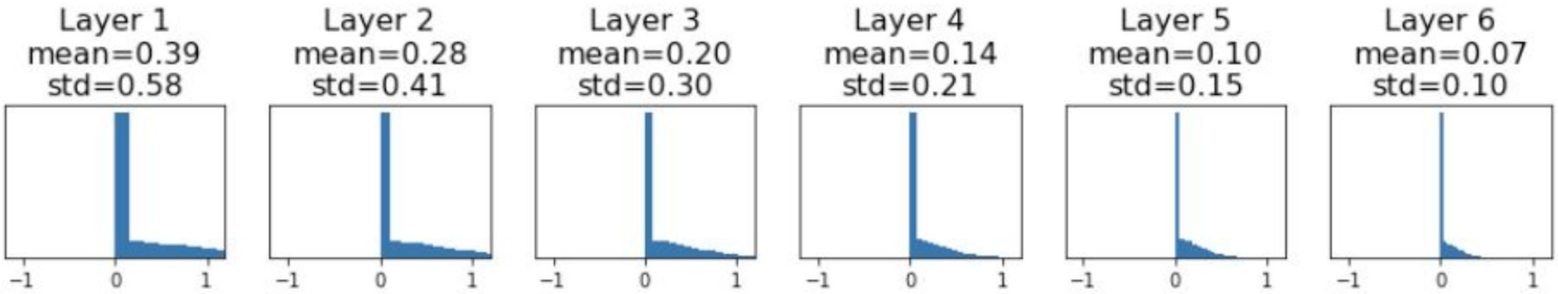
Xavier Initialization with ReLU

```
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 =(



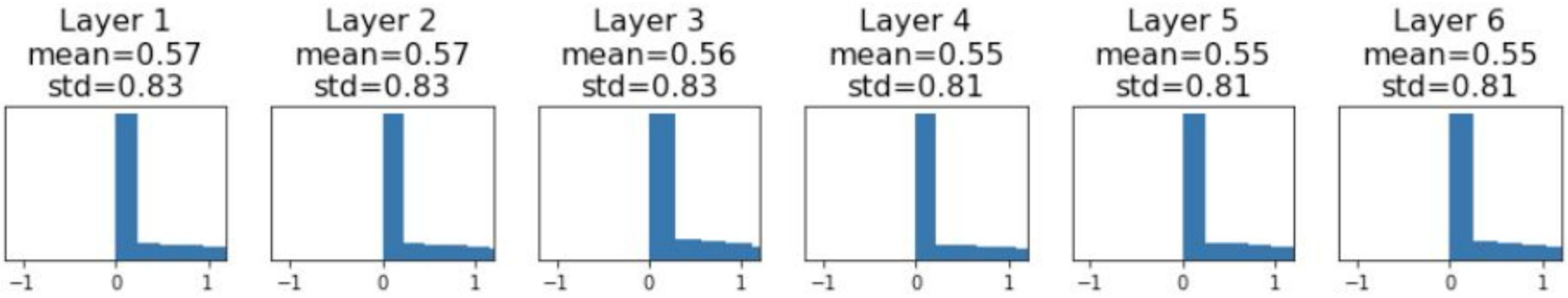
Slide credit: Stanford CS231n
Image credit: Stanford CS231n
Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, He et al., 2016

Kaiming 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!



Slide credit: Stanford CS231n
Image credit: Stanford CS231n
Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, He et al., 2016

Practical tips

- **Initialization matters**
- Large init is bad
- Small init is bad
- Xavier and Kaiming provide a good heuristic to get “just right” init range
- For very large networks (> 100M parameters) normal distribution produces too many outliers (very large numbers) which can heart convergence
  - Using uniform distribution instead of normal helps

$$XavierNormal(d_{in}) = \sqrt{\frac{1}{d_{in}}}N(0,1) \qquad XavierUniform(d_{in}) = \sqrt{\frac{6}{d_{in}}}U(0,1)$$

$$KaimingNormal(d_{in}) = \sqrt{\frac{2}{d_{in}}}N(0,1) \qquad KaimingUniform(d_{in}) = \sqrt{\frac{3}{d_{in}}}N(0,1)$$