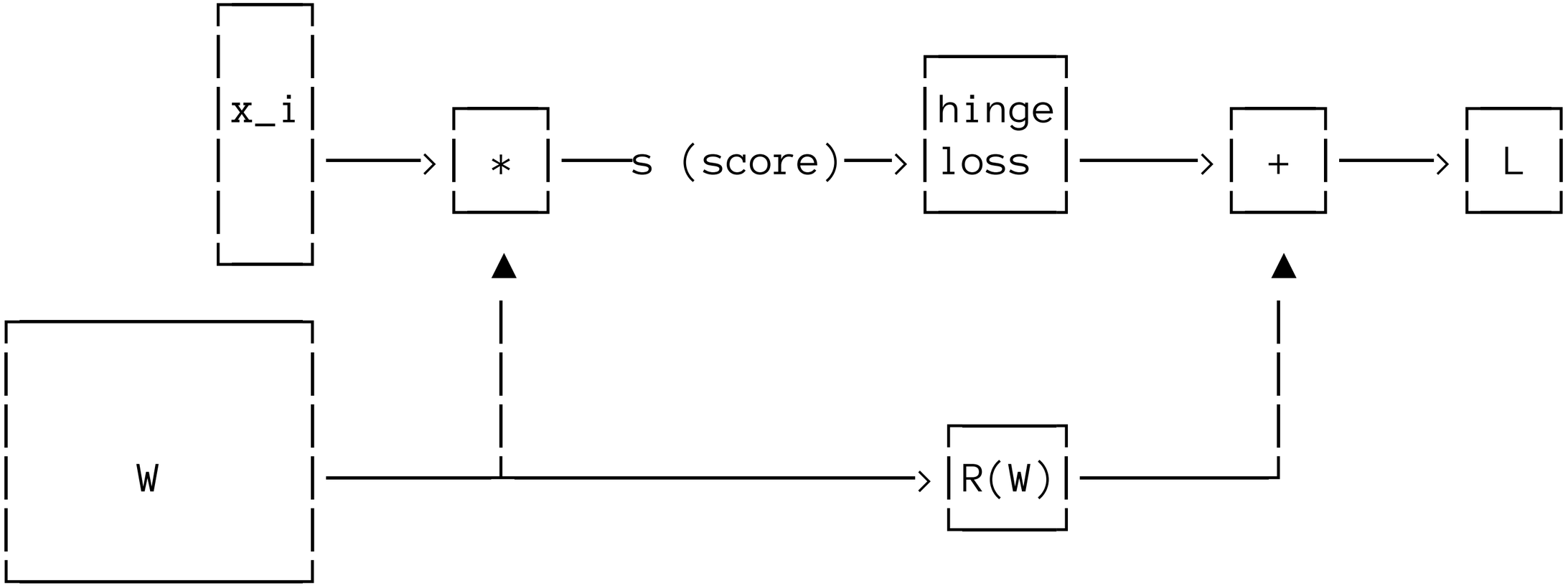
Deep Learning Bootcamp Day 2

Dr. Kashif Rasul[1]

25.09.2018 at Center for Systems Biology Dresden

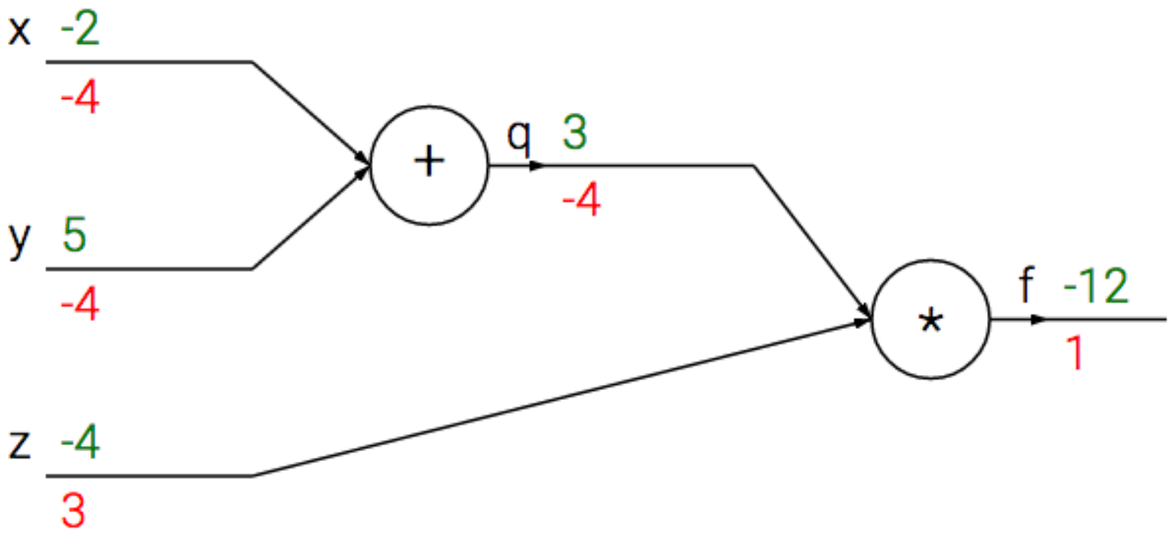
[1] Zalando Research Twitter: @krasul Github: kashif

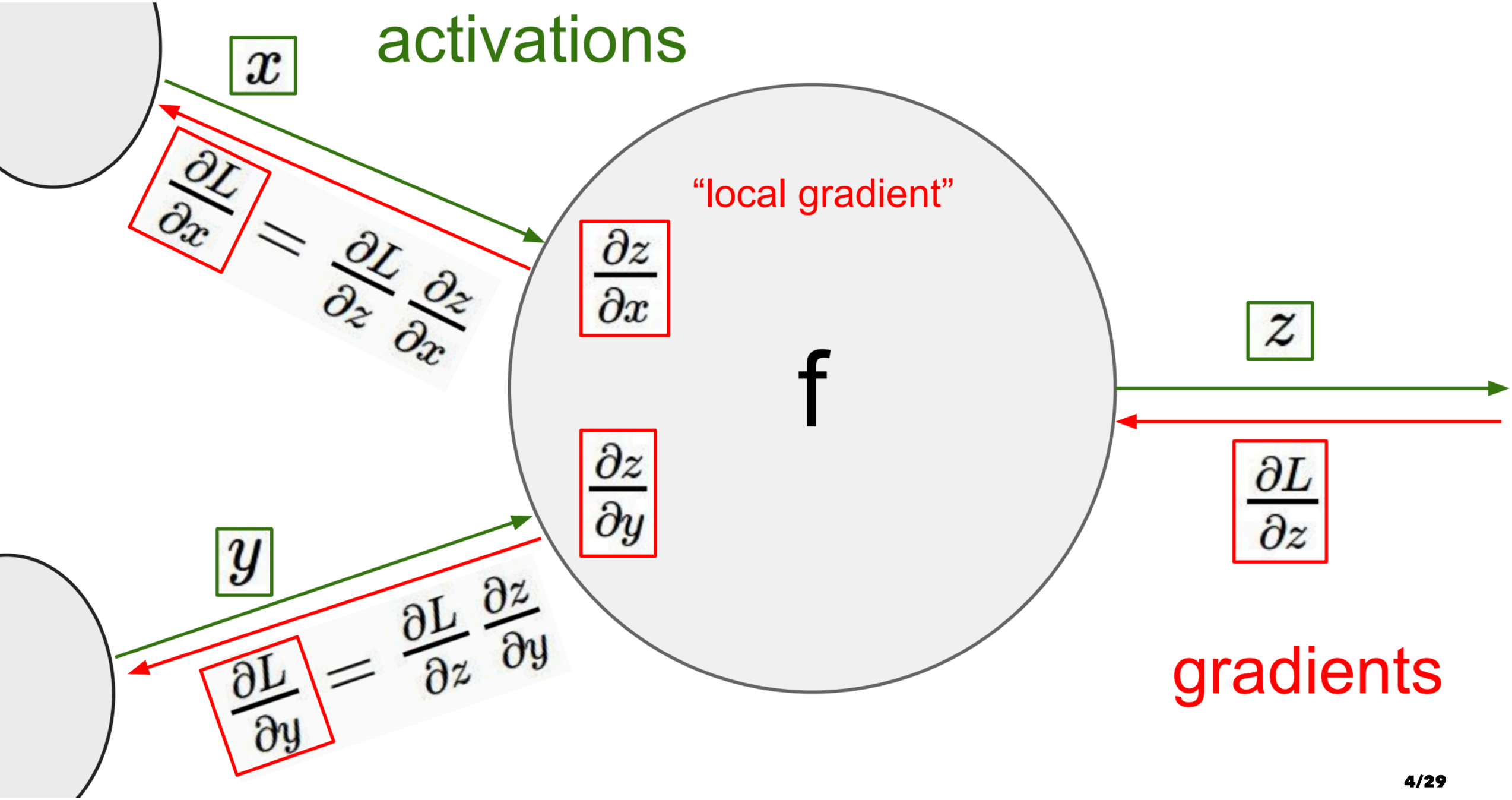
Computational Graph



Compound expression

- $f(x, y, z) = (x + y)z$
- E.g. x = −2, y = 5, z = −4
- $q = x + y$, $\frac{\partial q}{\partial x} = 1$, $\frac{\partial q}{\partial y} = 1$
- $f = qz$, $\frac{\partial f}{\partial q} = z$, $\frac{\partial f}{\partial z} = q$
- **Want**: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$
- **Chain rule**: $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$



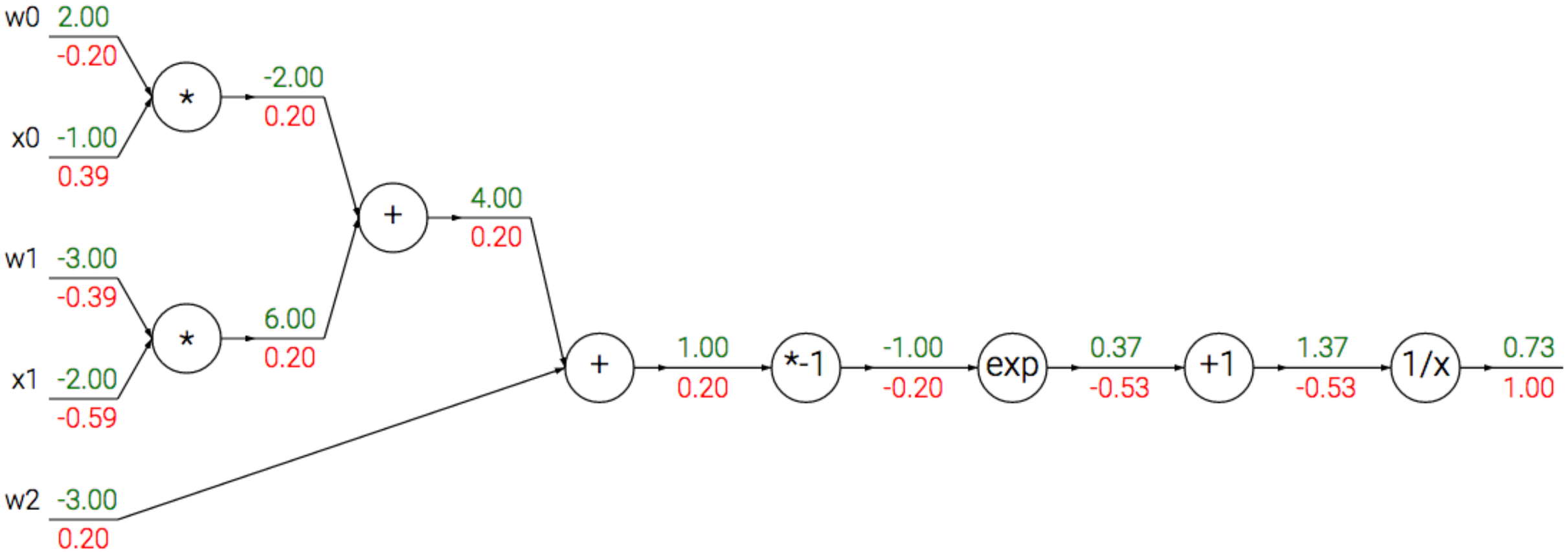

activations
x
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
"local gradient"
$\frac{\partial z}{\partial x}$
f
$\frac{\partial z}{\partial y}$
z
$\frac{\partial L}{\partial z}$
y
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$
gradients

Modularity

- Any differentiable function can act as a gate
- Can group multiple gates into a single gate
- Decompose a function into multiple gates

$$f(\vec{w}, \vec{x}) = \frac{1}{1 + \exp(-(w_0 x_0 + w_1 x_1 + w_2))}$$

- We have these new gates:
  - $f(x) = 1/x$ and $df/dx = -1/x^2$
  - $f(x) = c + x$ and $df/dx = 1$
  - $f(x) = \exp(x)$ and $df/dx = \exp(x)$
  - $f(x) = ax$ and $df/dx = a$


w0
2.00
-0.20
x0
-1.00
0.39
*
-2.00
0.20
w1
-3.00
-0.39
x1
-2.00
-0.59
*
6.00
0.20
+
4.00
0.20
w2
-3.00
0.20
+
1.00
0.20
*-1
-1.00
-0.20
exp
0.37
-0.53
+1
1.37
-0.53
1/x
0.73
1.00

Sigmoid function

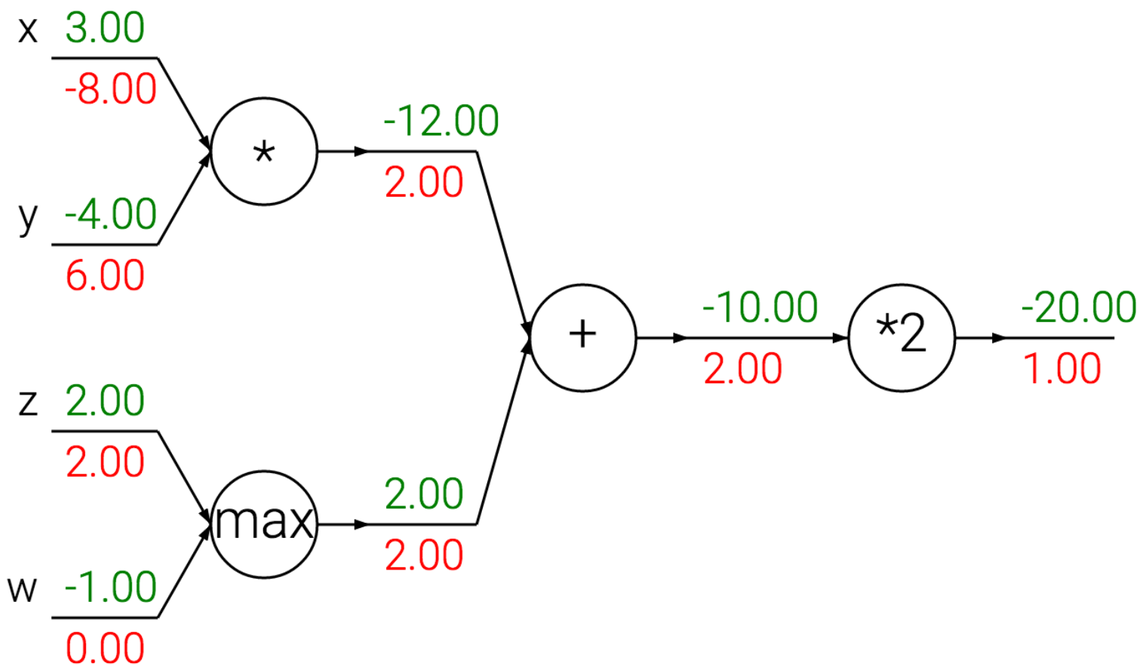
$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

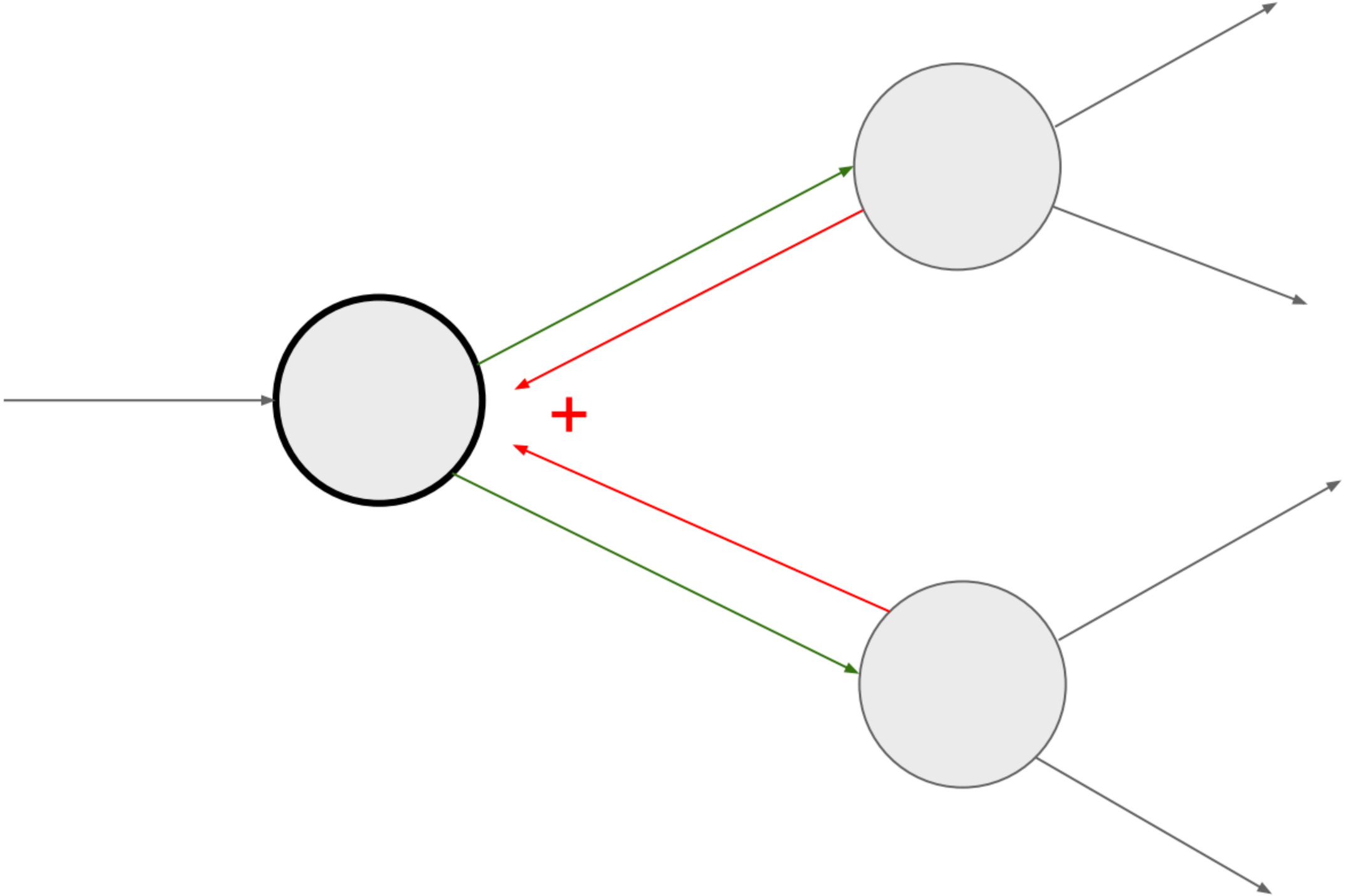
for which

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = \sigma(x)(1 - \sigma(x))$$

Patterns

- Add gate: gradient distributer
- Max gate: gradient router
- Multiply gate: gradient "switcher"





Unintuitive effects

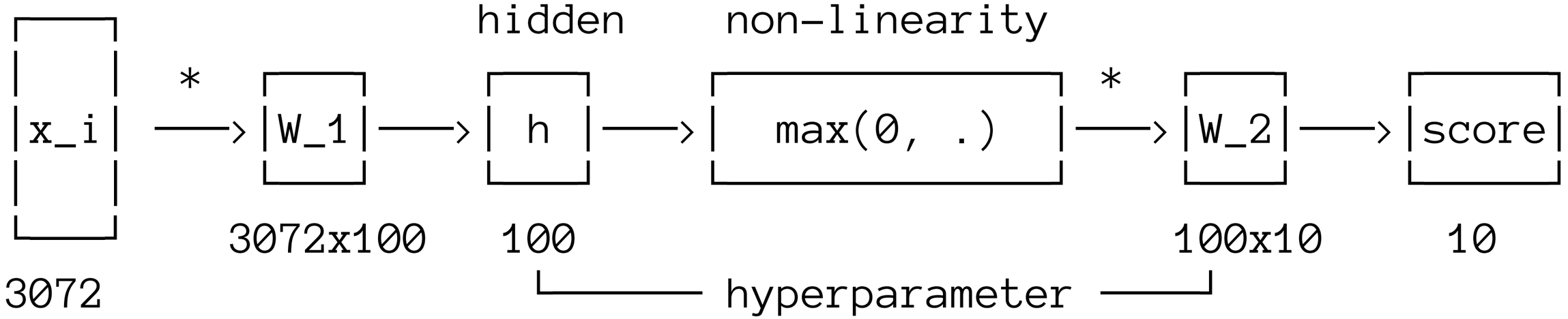
- If one of the inputs to the multiply gate is very small and the other is very big, then multiply gate will do something unintuitive
- It will assign a relatively huge gradient to the small input and a tiny gradient to the large input

Gradients of vectorized operators

- All concepts extend in a straight-forward manner to matrix and vector operations
- One must pay closer attention to dimensions and transpose operations
- Matrix-Matrix multiply gradient: most tricky operation

Neural Networks

- Linear score: $f = W\vec{x}$
- 2-Layer Neural Network: $f = W_2 \max(0, W_1 \vec{x})$

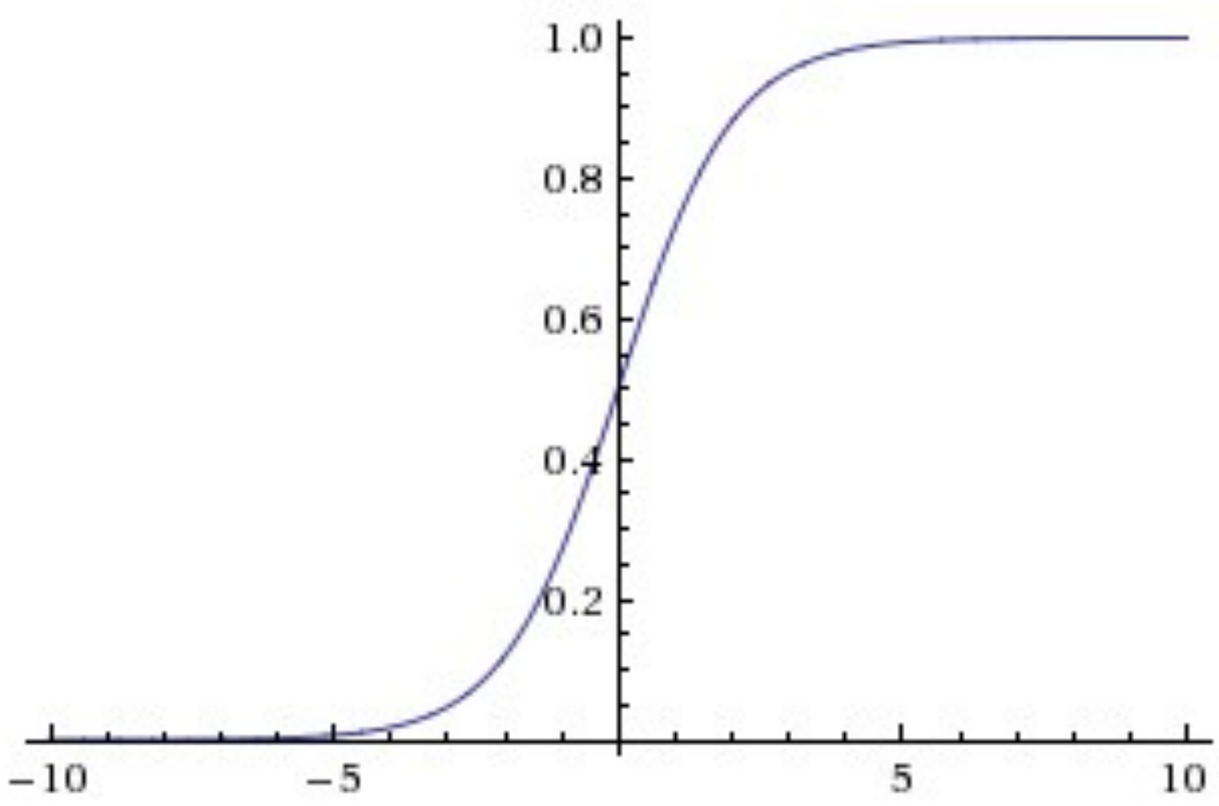


Activation functions

- Sigmoid: $\sigma(x) = 1/(1 + \mathbf{exp}(-x))$
- Tanh: $\mathbf{tanh}(x)$
- ReLU (hinge): $\mathbf{max}(0, x)$
- Leaky ReLU: $\mathbf{max}(0.1x, x)$
- Others: Maxout, PReLU, ELU, etc. (active area of research)

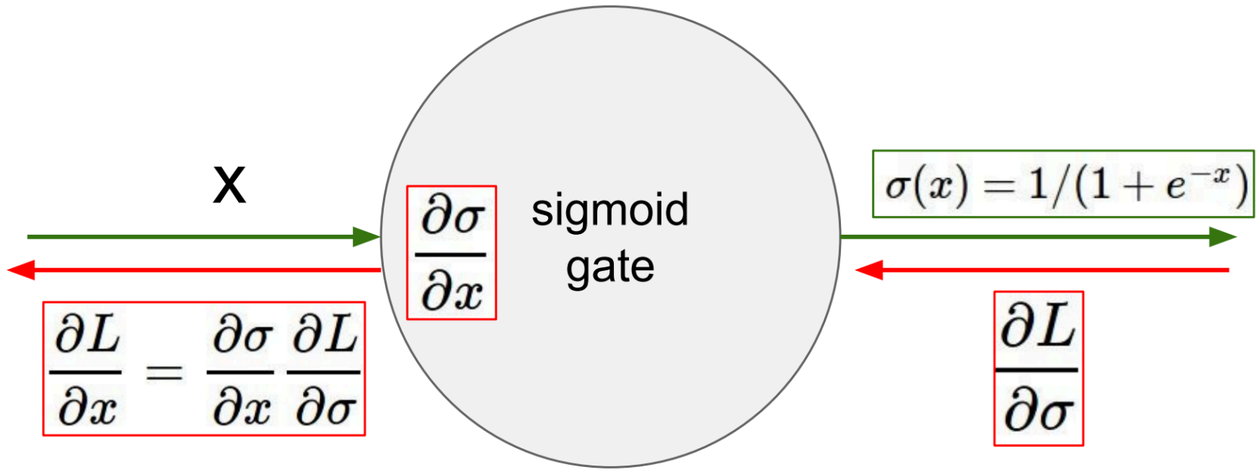
Sigmoid activation

- $\sigma(x) = 1/(1 + \exp(-x))$
- Squashes numbers to the range $(0, 1)$
- Issues:
  1. Saturated neurons "kill" the gradient
  2. Outputs not zero centered
  3. exp( ) computationally expensive



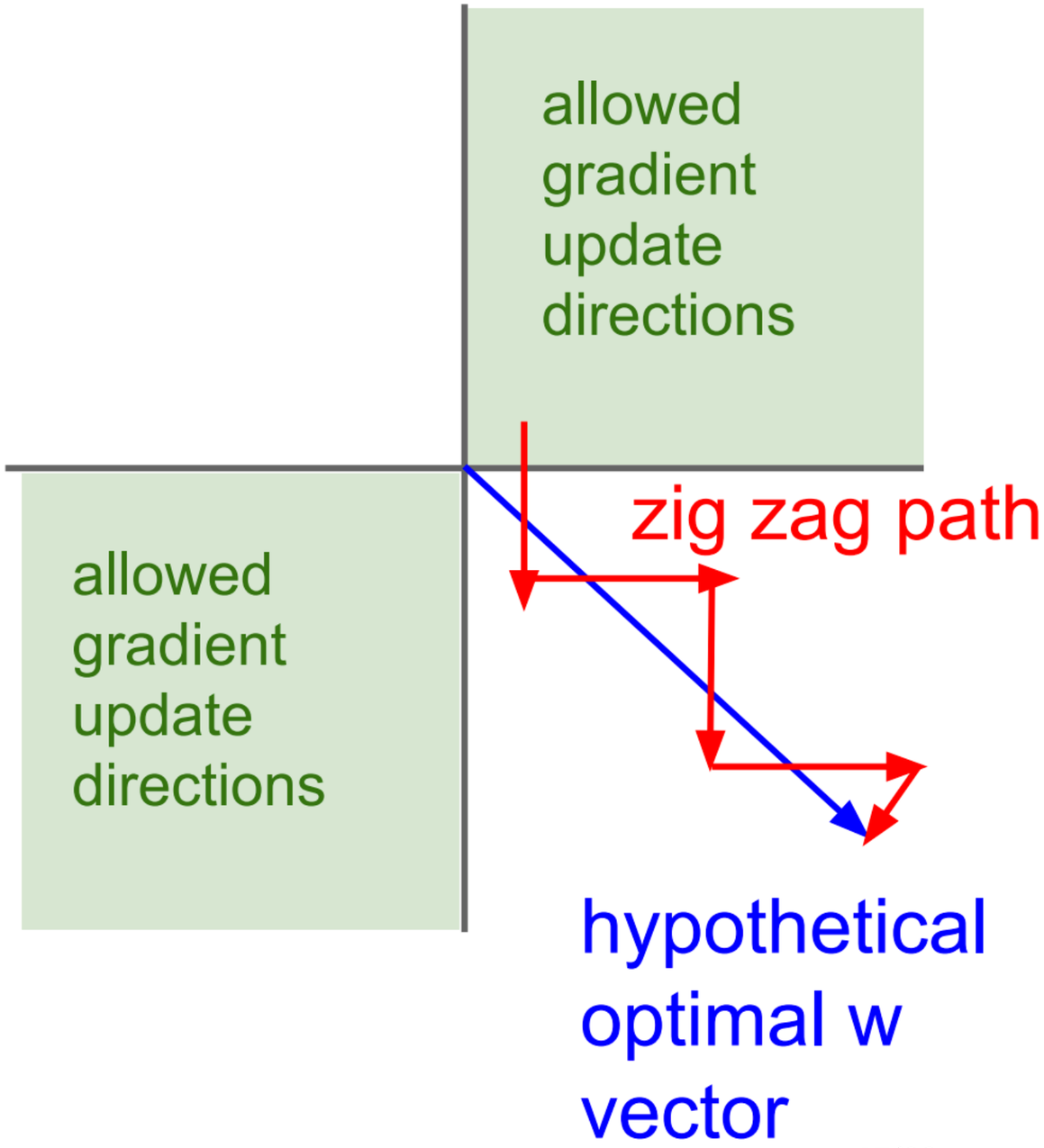
1. Saturation

- $\partial\sigma/\partial x = \sigma(x)(1-\sigma(x))$
- What happens to backward gradient when:
  - $x = -10$?
  - $x = 0$?
  - $x = 10$?



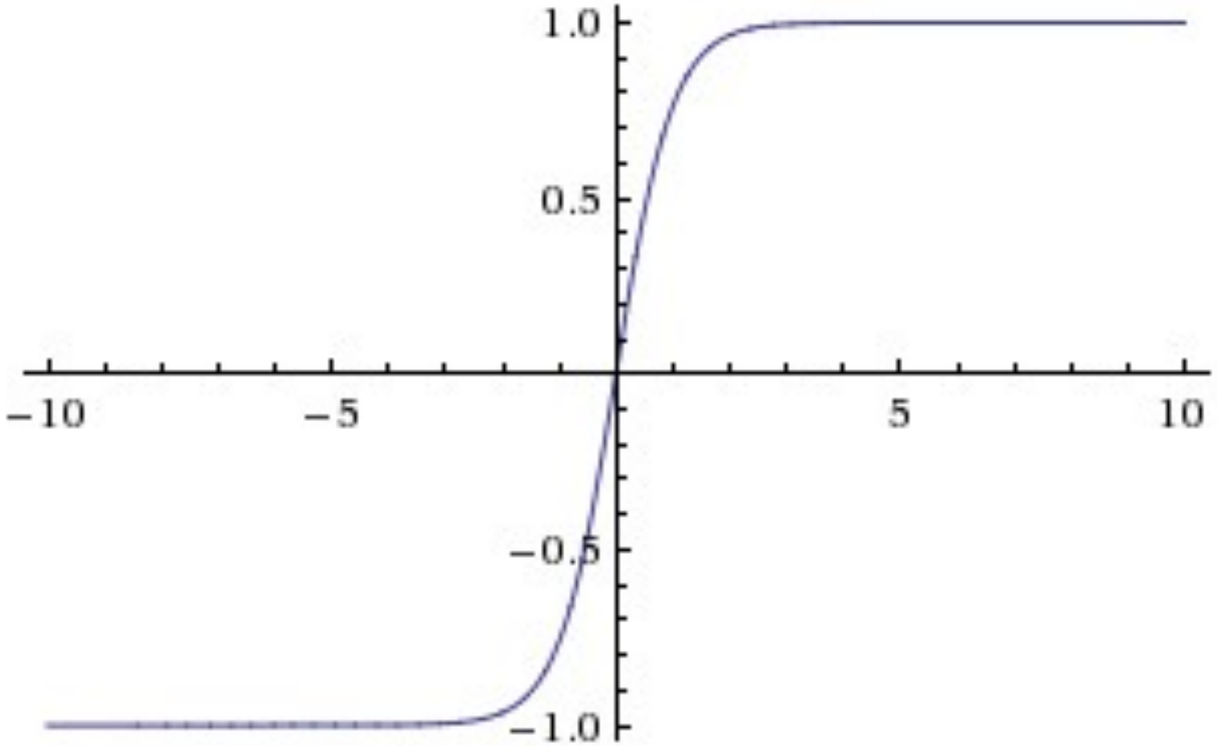
2. Positive output

- Suppose $f = \sum_i w_i x_i + b$, $x_i > 0$
- $\frac{\partial f}{\partial w_i} = x_i$
- $\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial w_i} = x_i \frac{\partial L}{\partial f}$
- So all the backward gradients **always** have the same sign as $\frac{\partial L}{\partial f}$



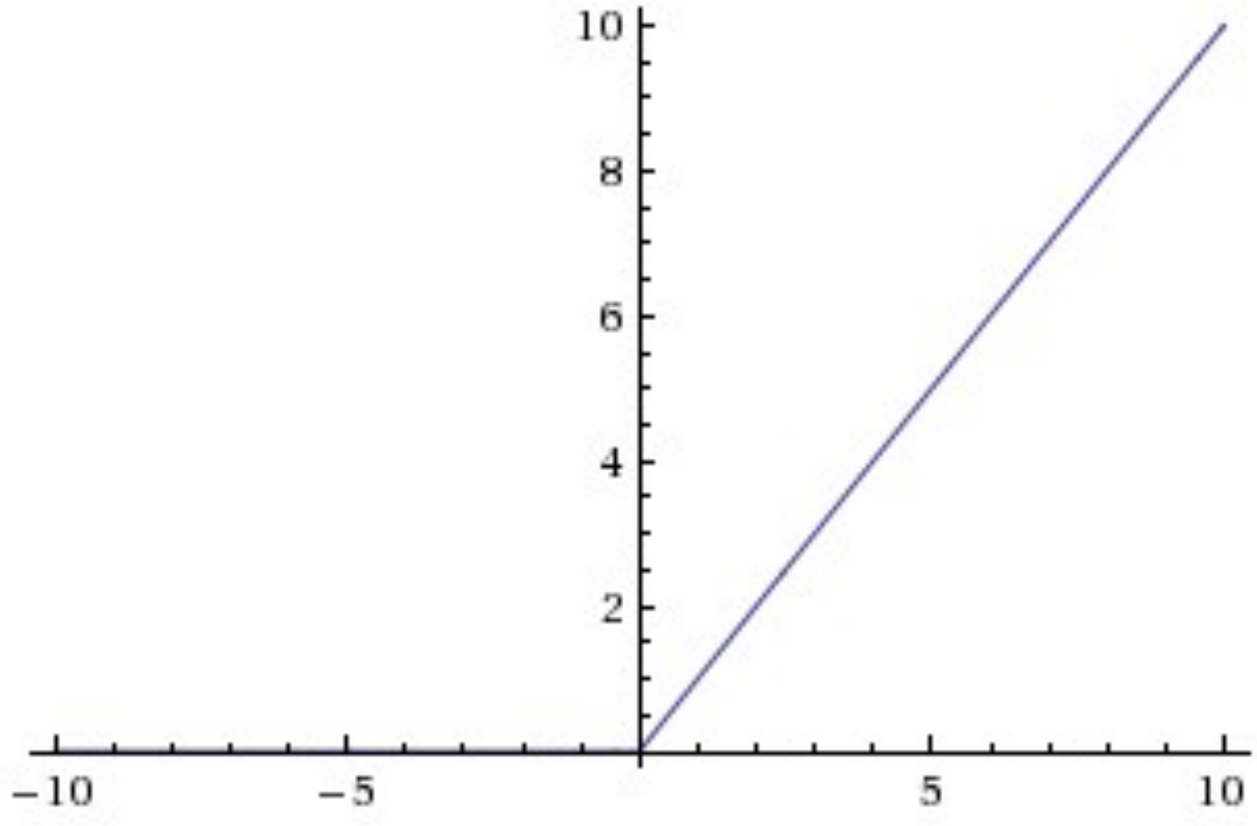
Tanh

- Squashes numbers to $(-1, 1)$
- Zero centered
- Still kills gradients when saturated



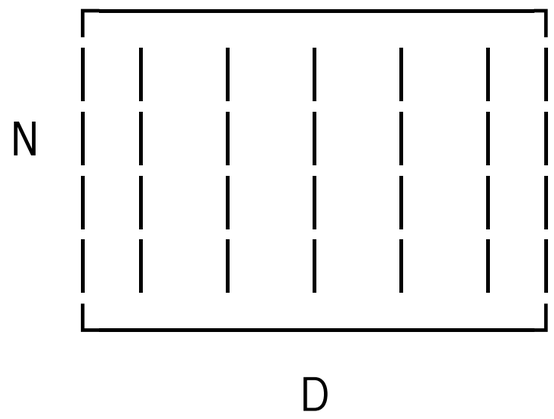
ReLU (Rectified Linear Unit)

- $f(x) = \max(0, x)$
- Does not saturate
- Very computationally efficient
- Converges faster than sigmoid or tanh in practice
- Issues:
  - Non-zero centered
  - Not smooth



Data Preprocessing: `X ([N,D])`

- Mean subtraction: `X -= np.mean(X, axis=0)`
- Normalization: `X /= np.std(X, axis=0)`



- PCA: decorrelates the data
- Whitening

Preprocessing for images: center only

- Subtract the mean image: e.g. CIFAR-10 needs a `[32,32,3]` array
- Subtract per-channel mean: 3 numbers, e.g. with `[N, C, W, H]` format

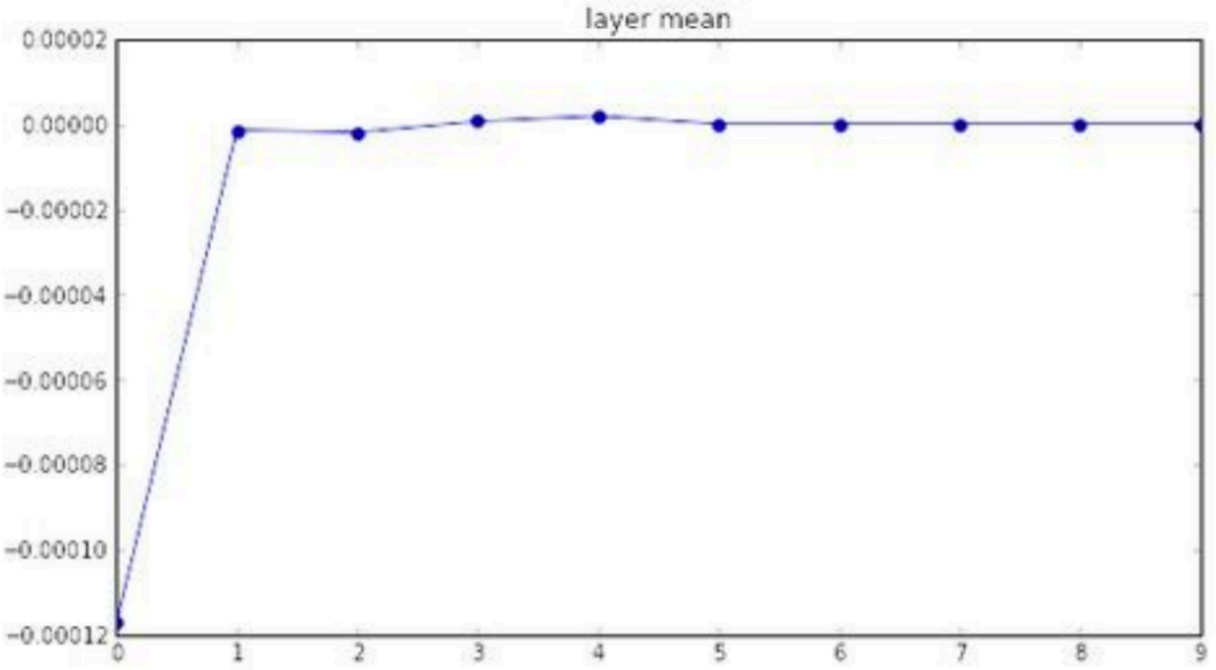
```
X[:, 0, :, :] -= 103.939 # R mean
X[:, 1, :, :] -= 116.779 # G mean
X[:, 2, :, :] -= 123.68  # B mean
```

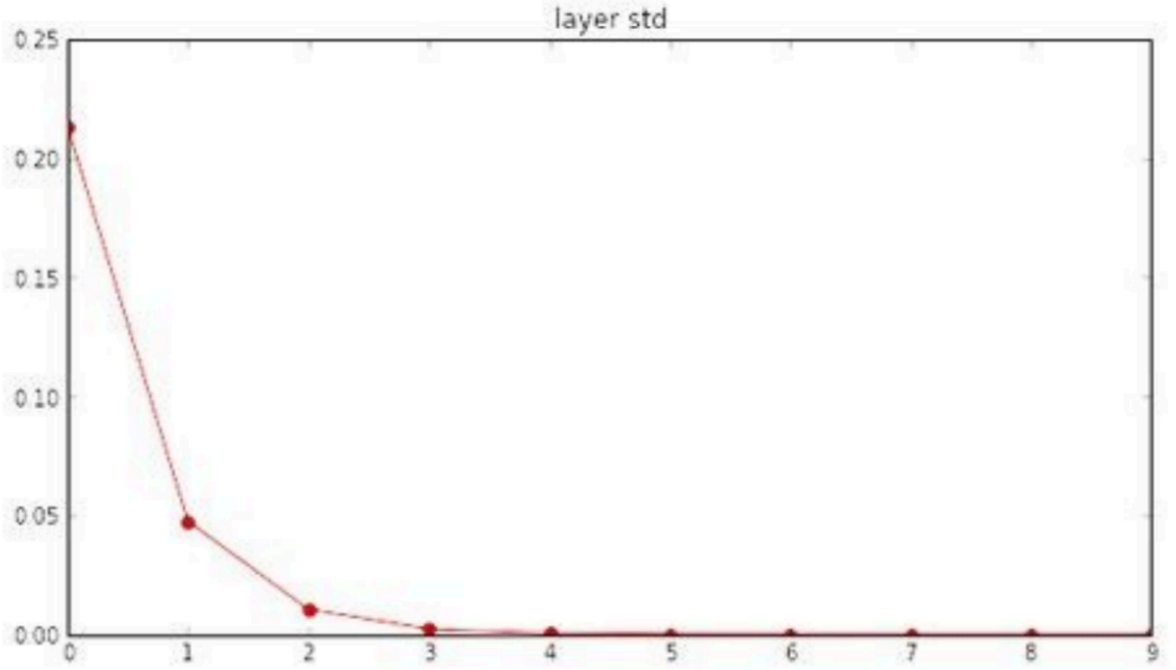
Weight initialization

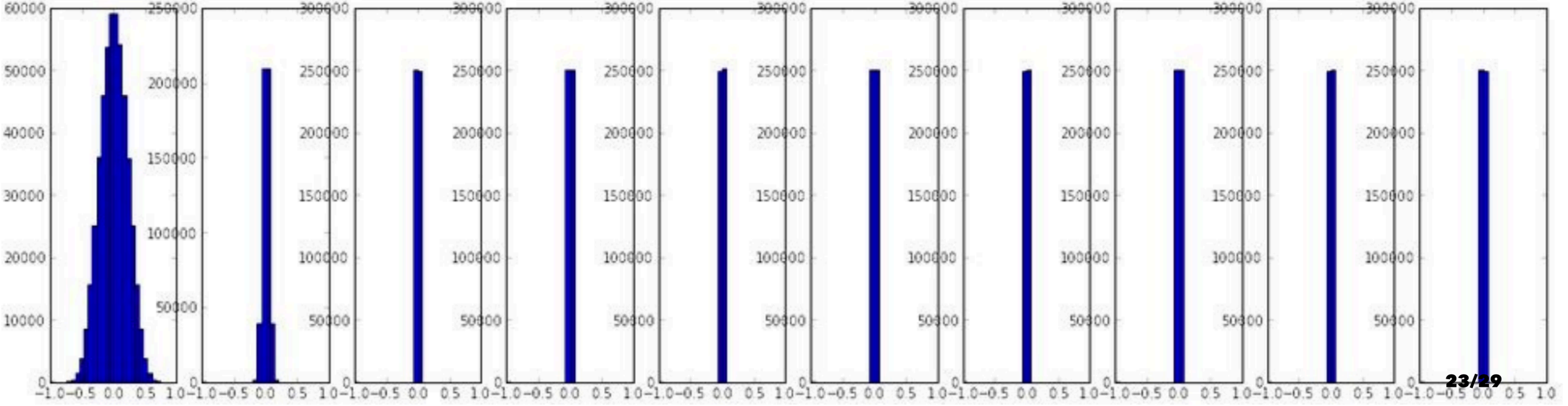
- What happens when $W = 0$ initialization is used?
- First idea: small random numbers

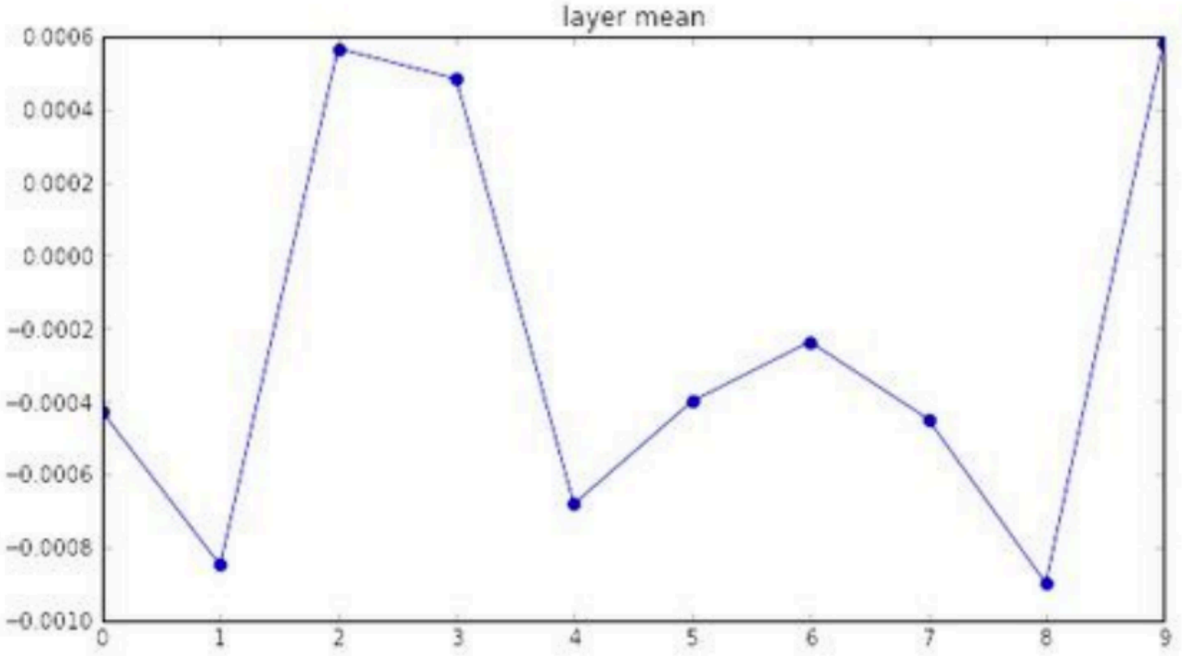
```
W = 0.01 * np.random.randn(D, H)
```

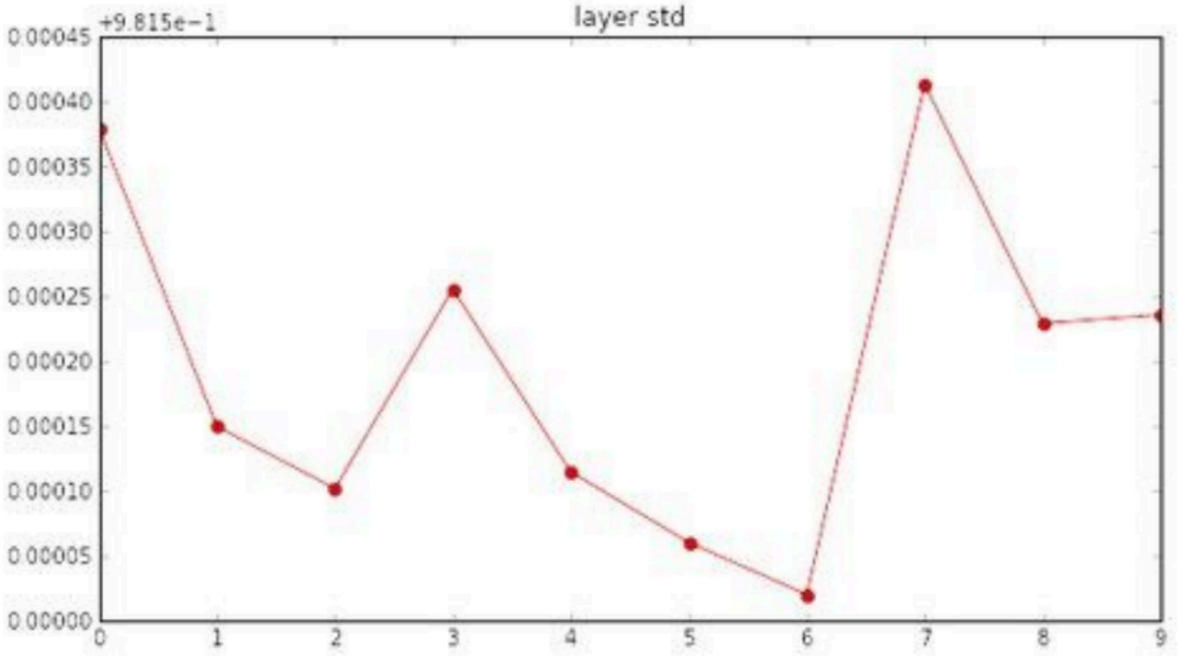
- To see the behavior need activation statistics:
  - mean and variance of values of the weight matrix
  - histogram of values of weight matrix

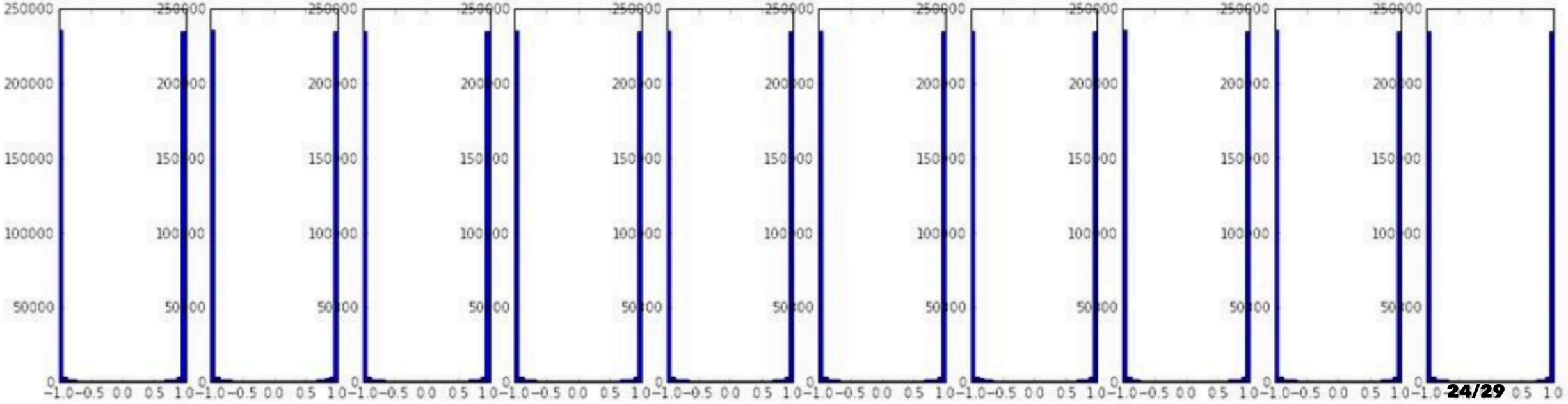

layer mean
0.00002
0.00000
−0.00002
−0.00004
−0.00006
−0.00008
−0.00010
−0.00012
0 1 2 3 4 5 6 7 8 9


layer std
0.25
0.20
0.15
0.10
0.05
0.00
0 1 2 3 4 5 6 7 8 9


60000
50000
40000
30000
20000
10000
0
250000
200000
150000
100000
50000
300000
−1.0 −0.5 0.0 0.5 1.0


layer mean
0.0006
0.0004
0.0002
0.0000
−0.0002
−0.0004
−0.0006
−0.0008
−0.0010
0 1 2 3 4 5 6 7 8 9


layer std
+9.815e−1
0.00045
0.00040
0.00035
0.00030
0.00025
0.00020
0.00015
0.00010
0.00005
0.00000
0 1 2 3 4 5 6 7 8 9


250000
200000
150000
100000
50000
0
−1.0 −0.5 0.0 0.5 1.0

Nice weights?

- For $f_j = W_j \vec{x} = \sum_{i=1}^{n_{\text{in}}} w_{ji} x_i$ we have:

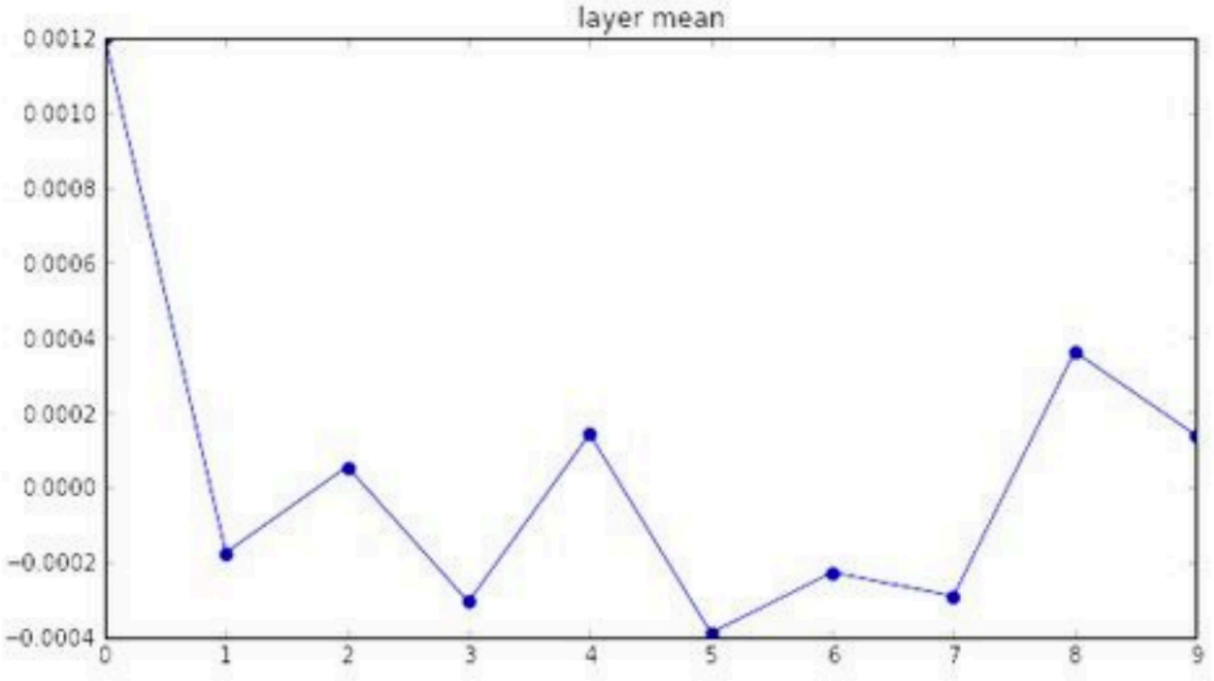
  $\mathbf{Var}(w_{ji} x_i) = [\mathbb{E}(x_i)]^2 \mathbf{Var}(w_{ji}) + [\mathbb{E}(w_{ji})]^2 \mathbf{Var}(x_i) + \mathbf{Var}(x_i)\mathbf{Var}(w_{ji})$
- If x_i and w_{ji} have mean 0 and are independent random then $\mathbf{Var}(f_j) = n_{\text{in}} \mathbf{Var}(x_i)\mathbf{Var}(w_{ji})$
- For the variance of input and output to be the same set $\mathbf{Var}(w_{ji}) = 1/n_{\text{in}}$ for all i and j

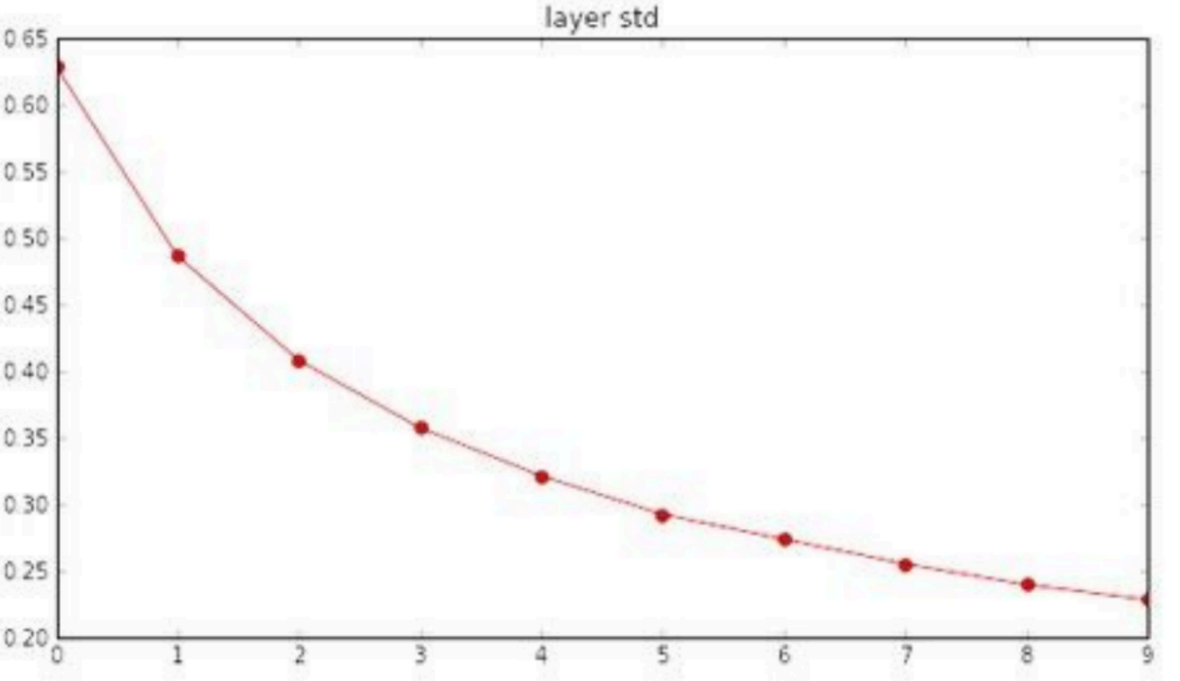
- Gradient to have the same variance: $\mathbf{Var}(w_{ji}) = 1/n_{\text{out}}$
- Compromise: $\mathbf{Var}(W) = 2/(n_{\text{in}} + n_{\text{out}})$

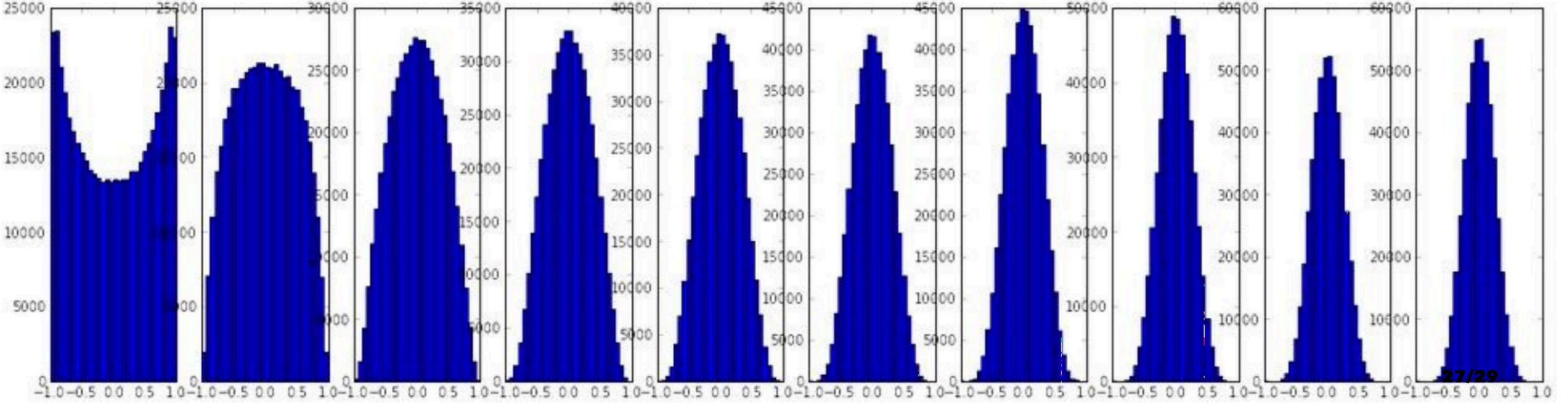
```
from keras import initializers

model.add(Dense(64,
  kernel_initializer=initializers.glorot_normal()))

# also works; will use the default parameters.
model.add(Dense(64, kernel_initializer='glorot_normal'))
```


layer mean
0.0012
0.0010
0.0008
0.0006
0.0004
0.0002
0.0000
−0.0002
−0.0004
0 1 2 3 4 5 6 7 8 9


layer std
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0 1 2 3 4 5 6 7 8 9


25000
20000
15000
10000
5000
0
−1.0 −0.5 0.0 0.5 1.0

Delving Deep into Rectifiers

- Instead of identity consider ReLU non-linearites
- Then: $\mathbf{Var}(f_i) = n\mathbb{E}(x_i^2)\mathbf{Var}(W_i)$ since mean of $\mathbf{max}(0, x_i)$ is not 0
- So for variance of input and output to be the same we need: $\mathbf{Var}(W) = \mathbf{2}/n_{\mathbf{in}}$

```
model.add(Dense(64, kernel_initializer='he_normal'))
```

Optimal performance?*

- May be using the wrong criteria
- The weight properties do not persist as learning starts
- Learning speed increases but inadvertently so does the generalization error
- For large layers (big n_{in}), variance of $2/n_{\text{in}}$ means extremely small weights

---

* Deep Learning Book (2016)