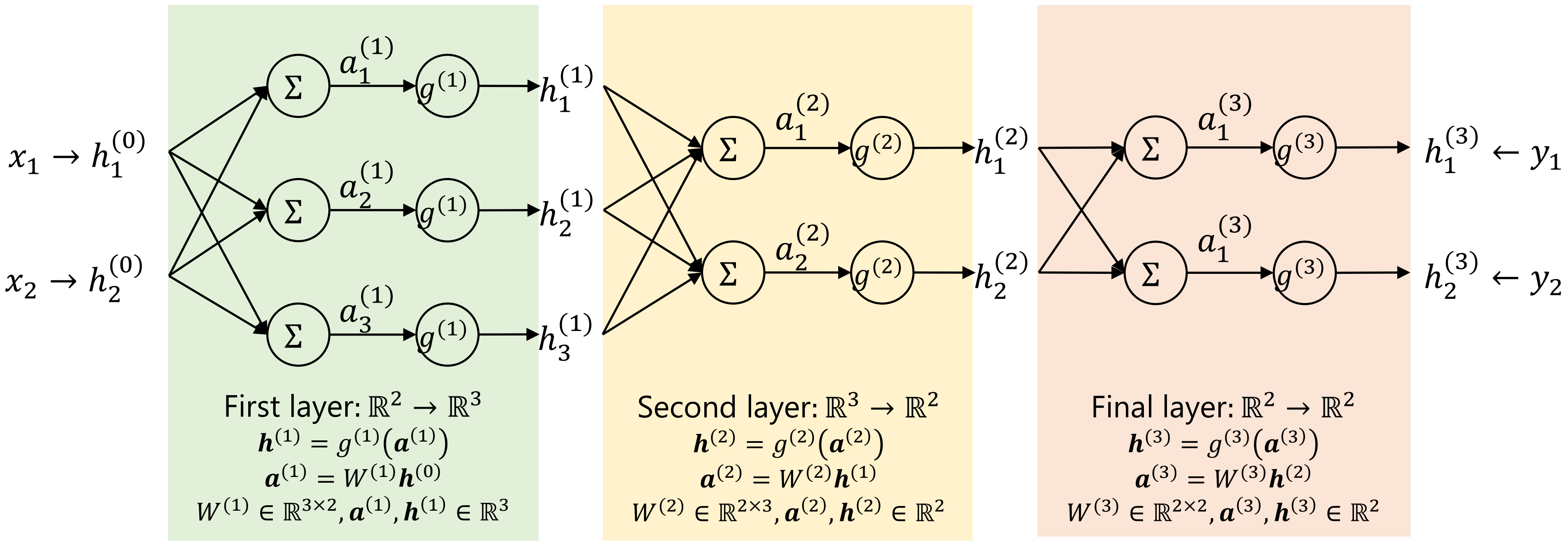
Generic form of Feedforward Neural Networks

Designing feedforward neural networks



First layer: $\mathbb{R}^2 \rightarrow \mathbb{R}^3$
$\boldsymbol{h}^{(1)} = g^{(1)}\left(\boldsymbol{a}^{(1)}\right)$
$\boldsymbol{a}^{(1)} = W^{(1)}\boldsymbol{h}^{(0)}$
$W^{(1)} \in \mathbb{R}^{3\times 2}, \boldsymbol{a}^{(1)}, \boldsymbol{h}^{(1)} \in \mathbb{R}^3$

Second layer: $\mathbb{R}^3 \rightarrow \mathbb{R}^2$
$\boldsymbol{h}^{(2)} = g^{(2)}\left(\boldsymbol{a}^{(2)}\right)$
$\boldsymbol{a}^{(2)} = W^{(2)}\boldsymbol{h}^{(1)}$
$W^{(2)} \in \mathbb{R}^{2\times 3}, \boldsymbol{a}^{(2)}, \boldsymbol{h}^{(2)} \in \mathbb{R}^2$

Final layer: $\mathbb{R}^2 \rightarrow \mathbb{R}^2$
$\boldsymbol{h}^{(3)} = g^{(3)}\left(\boldsymbol{a}^{(3)}\right)$
$\boldsymbol{a}^{(3)} = W^{(3)}\boldsymbol{h}^{(2)}$
$W^{(3)} \in \mathbb{R}^{2\times 2}, \boldsymbol{a}^{(3)}, \boldsymbol{h}^{(3)} \in \mathbb{R}^2$

- The number of layers
- The numbers of dimensions of hidden layers
- An activation function for each layer
- A loss function

Cross entropy loss

- For binary classification

$$l(a, y) = -y \log \sigma(a) - (1 - y) \log\big(1 - \sigma(a)\big)$$

- For multi-class classification

$$l(\boldsymbol{a}, y) = -a_y + \log \sum_k \exp(a_k)$$

- Cross entropy

True probability distribution
(1 for true category; 0 otherwise)

$$H(p, q) = -\sum_k p(k) \log q(k)$$

Predicted probability distribution

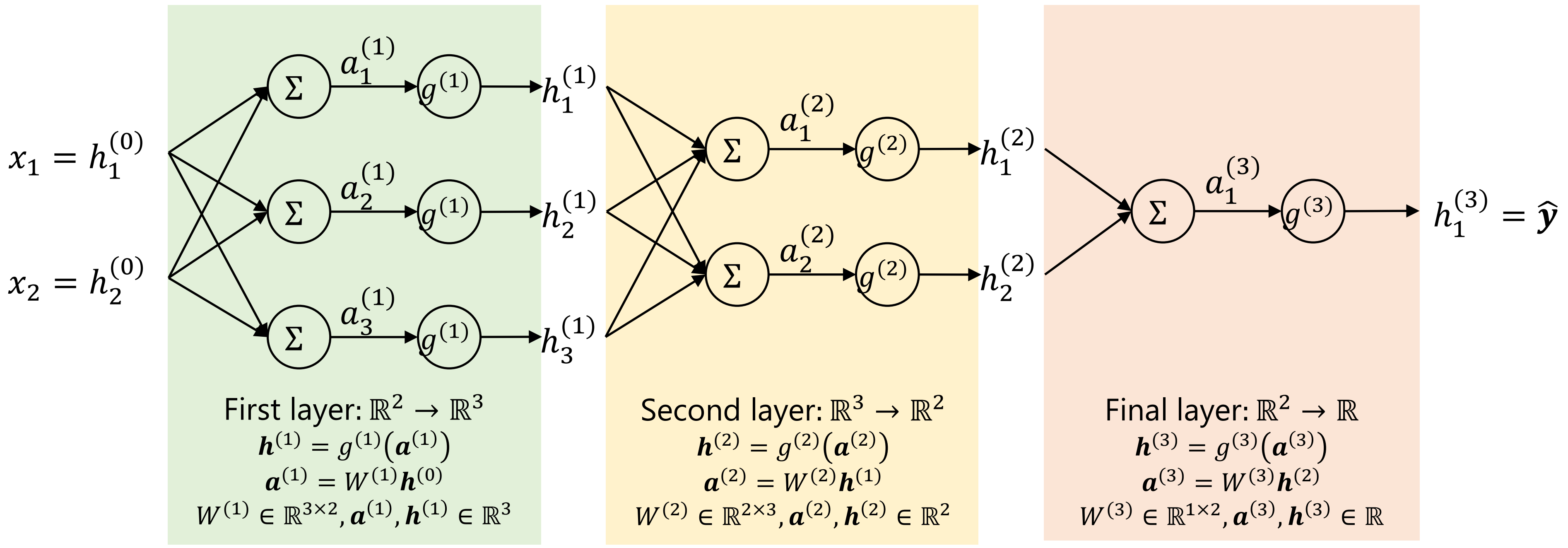
Mean Squared Error (MSE) loss

- Used for regression

$$l(\boldsymbol{a}, \boldsymbol{y}) = \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{a}\|_2^2$$

Training multi-layer neural networks and back propagation

Generic notation for multi-layer NNs



First layer: $\mathbb{R}^2 \to \mathbb{R}^3$
$\boldsymbol{h}^{(1)} = g^{(1)}\left(\boldsymbol{a}^{(1)}\right)$
$\boldsymbol{a}^{(1)} = W^{(1)}\boldsymbol{h}^{(0)}$
$W^{(1)} \in \mathbb{R}^{3\times 2}, \boldsymbol{a}^{(1)}, \boldsymbol{h}^{(1)} \in \mathbb{R}^3$

Second layer: $\mathbb{R}^3 \to \mathbb{R}^2$
$\boldsymbol{h}^{(2)} = g^{(2)}\left(\boldsymbol{a}^{(2)}\right)$
$\boldsymbol{a}^{(2)} = W^{(2)}\boldsymbol{h}^{(1)}$
$W^{(2)} \in \mathbb{R}^{2\times 3}, \boldsymbol{a}^{(2)}, \boldsymbol{h}^{(2)} \in \mathbb{R}^2$

Final layer: $\mathbb{R}^2 \to \mathbb{R}$
$\boldsymbol{h}^{(3)} = g^{(3)}\left(\boldsymbol{a}^{(3)}\right)$
$\boldsymbol{a}^{(3)} = W^{(3)}\boldsymbol{h}^{(2)}$
$W^{(3)} \in \mathbb{R}^{1\times 2}, \boldsymbol{a}^{(3)}, \boldsymbol{h}^{(3)} \in \mathbb{R}$

- The l–th layer ($l \in \{1, \dots, L\}$) consists of:
  - Input: $\boldsymbol{h}^{(l-1)} \in \mathbb{R}^{d_{l-1}}$ $\quad (\boldsymbol{h}^{(0)} = \boldsymbol{x})$
  - Output: $\boldsymbol{h}^{(l)} \in \mathbb{R}^{d_l}$ $\quad (\boldsymbol{h}^{(L)} = \hat{\boldsymbol{y}})$
  - Weight: $W^{(l)} \in \mathbb{R}^{d_l \times d_{l-1}}$
  - Activation function: $g^{(l)}$
  - Activation: $\boldsymbol{a}^{(l)} \in \mathbb{R}^{d_l}$

$$\boldsymbol{h}^{(l)} = g^{(l)}(W^{(l)}\boldsymbol{h}^{(l-1)})$$

$$W^{(l)} = \left(w_{ij}^{(l)}\right)$$

$w_{ij}^{(l)}$: weight from the j-th neuron to the i-th neuron of the l-th layer

Please accept the notational conflict between an instance-wise loss l_n and a layer number l

How to train weights in MLPs

- We have no explicit supervision signals for the internal (hidden) inputs/outputs $\boldsymbol{h}^{(2)}, \dots, \boldsymbol{h}^{(L-1)}$

- Having said that, SGD only needs the value of gradient $\frac{\partial l_n}{\partial w_{ij}^{(l)}}$ for every weight $w_{ij}^{(l)}$ in MLPs

- Can we compute the value of $\frac{\partial l_n}{\partial w_{ij}^{(l)}}$ for every weight $w_{ij}^{(l)}$?

- Yes! *Backpropagation* can do that!!

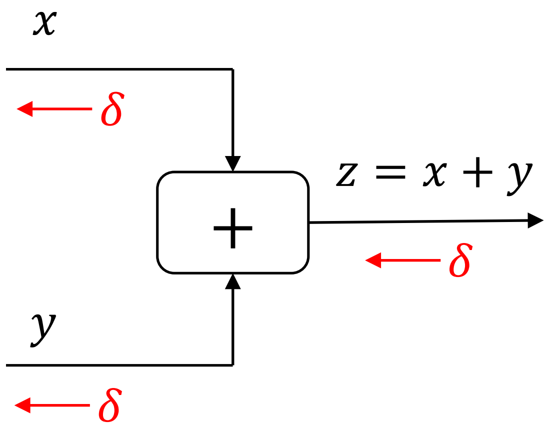
Backpropagation

- Commonly used in deep neural networks
- Formulas for backpropagation look complicated
- However:
  - We can understand backpropagation easily if we know the concept of *computation graph*
  - Most deep learning frameworks implement backpropagation by using *automatic differentiation*
- Let’s see computation graph and automatic differentiation first

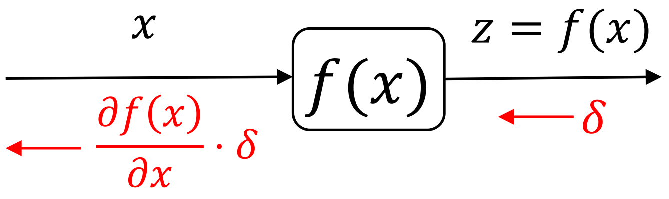
General Back-Propagation

The back-propagation algorithm is very simple. To compute the gradient of some scalar z with respect to one of its ancestors $\boldsymbol{x}$ in the graph, we begin by observing that the gradient with respect to z is given by $\frac{dz}{dz} = 1$. We can then compute the gradient with respect to each parent of z in the graph by multiplying the current gradient by the Jacobian of the operation that produced z. We continue multiplying by Jacobians traveling backwards through the graph in this way until we reach $\boldsymbol{x}$. For any node that may be reached by going backwards from z through two or more paths, we simply sum the gradients arriving from different paths at that node.

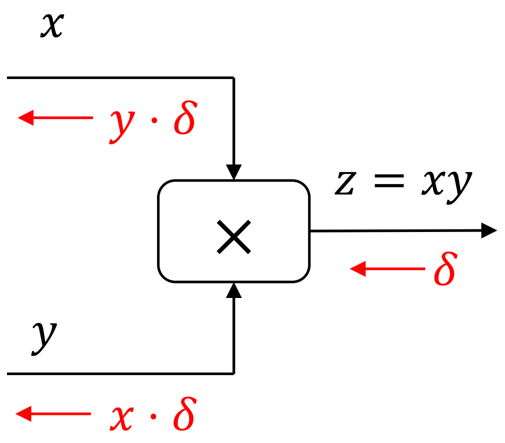
Rules for reverse-mode AD



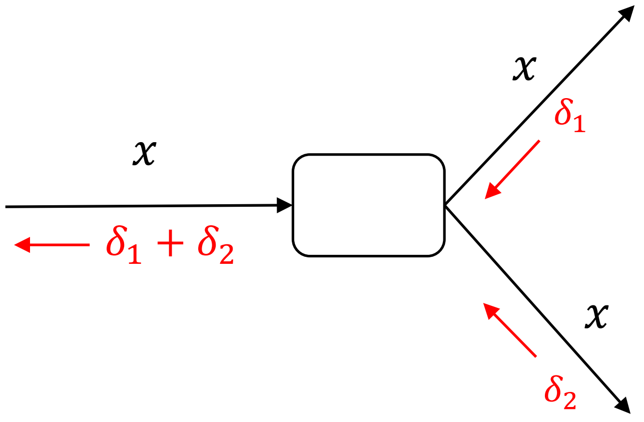
Add



Function application



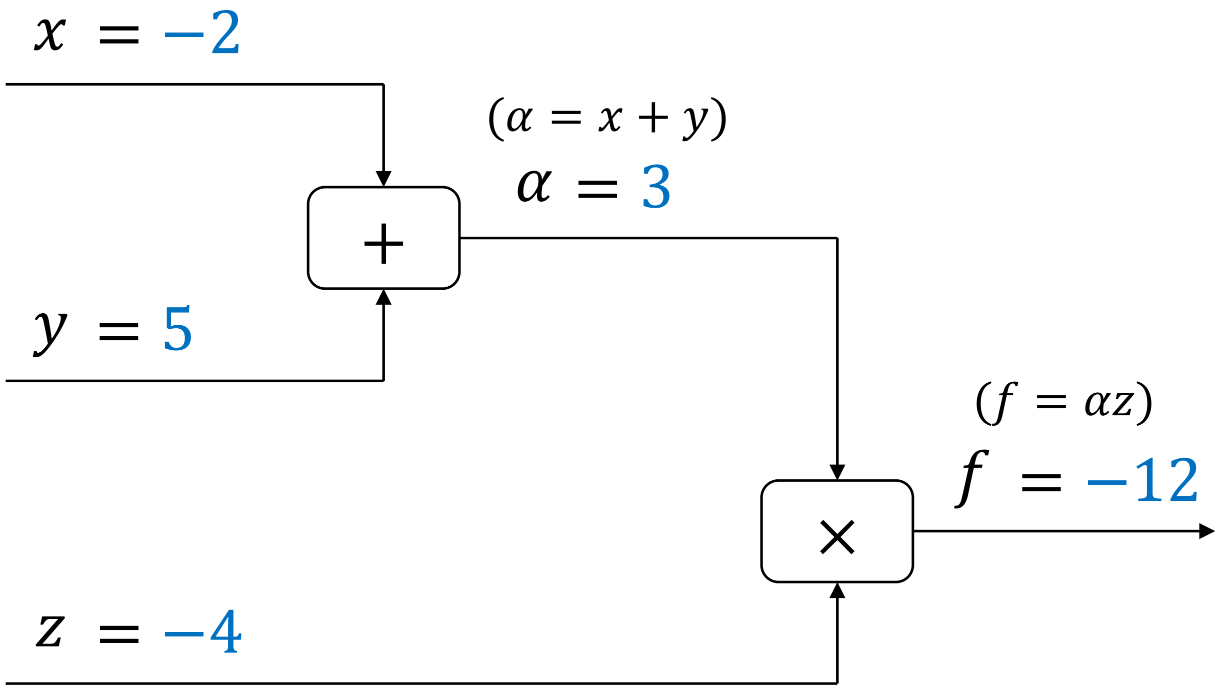
Multiply



Branch

Computation graph: $f(x, y, z) = (x + y)z$

http://cs231n.github.io/optimization-2/

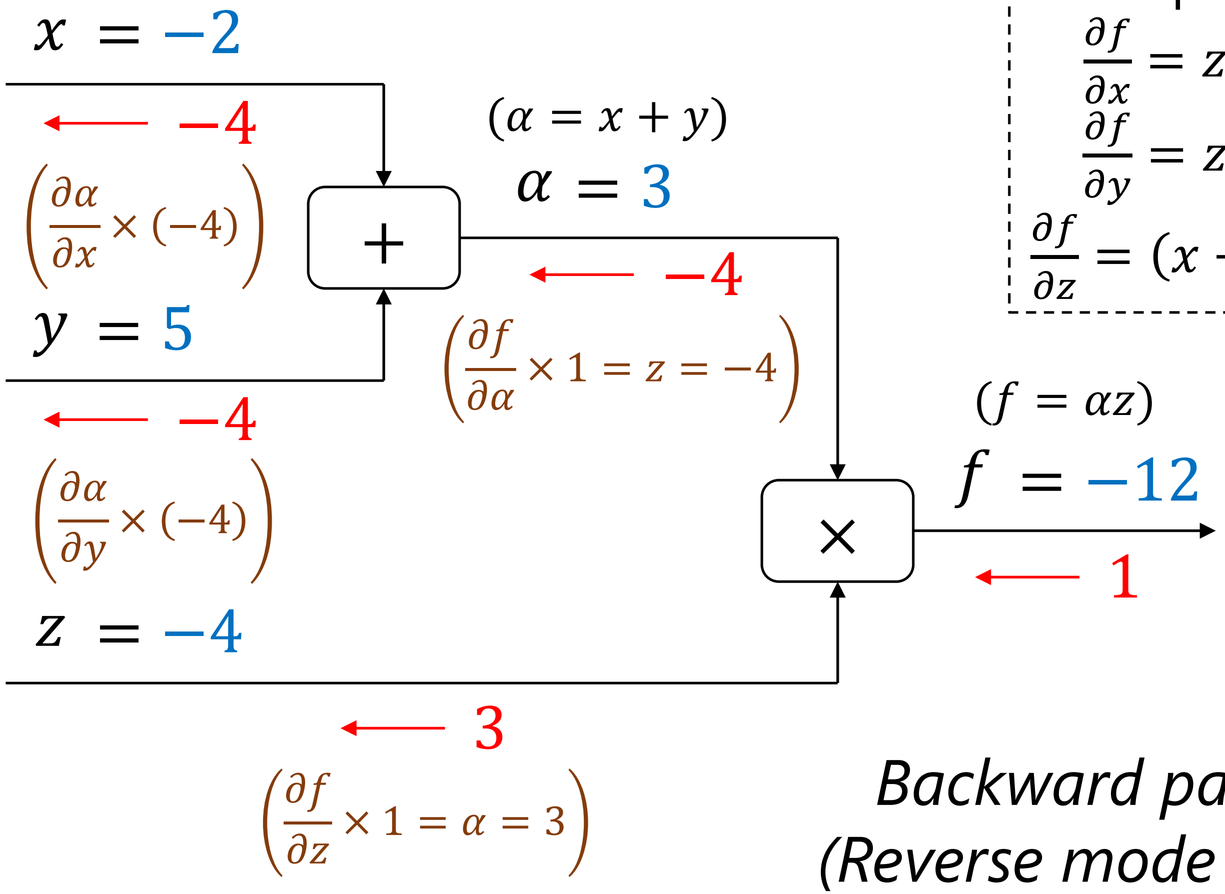


Forward pass

The value of a variable (above an arrow)

Automatic Differentiation (AD): $f(x, y, z) = (x + y)z$

http://cs231n.github.io/optimization-2/



$x = -2$
-4
$\left(\frac{\partial \alpha}{\partial x} \times (-4)\right)$
$y = 5$
-4
$\left(\frac{\partial \alpha}{\partial y} \times (-4)\right)$
$z = -4$
3
$\left(\frac{\partial f}{\partial z} \times 1 = \alpha = 3\right)$
$+$
$(\alpha = x + y)$
$\alpha = 3$
-4
$\left(\frac{\partial f}{\partial \alpha} \times 1 = z = -4\right)$
$\times$
$(f = \alpha z)$
$f = -12$
1

Compare with:

$$\frac{\partial f}{\partial x} = z = -4$$
$$\frac{\partial f}{\partial y} = z = -4$$
$$\frac{\partial f}{\partial z} = (x + y) = 3$$

Backward pass
(Reverse mode AD)

The value of a variable (above an arrow)
The gradient of the output f with respect to the variable (below an arrow)

Automatic differentiation (Baydin+ 2018)

- AD computes derivations by using the *chain rule*
  - Function values computed in the forward pass
  - Derivations computed with respect to:
    - Every variable (in reverse-mode accumulation)
    - A specific variable (in forward-mode accumulation)

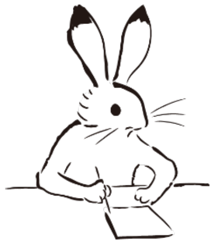
- Do not confuse with these:
  - Numerical differentiation: for example, $\frac{\partial f(x)}{\partial x} = \frac{f(x+\delta) - f(x)}{\delta}$
  - Symbolic differentiation: e.g., Mathematica, sympy

Exercise: AD on computation graph

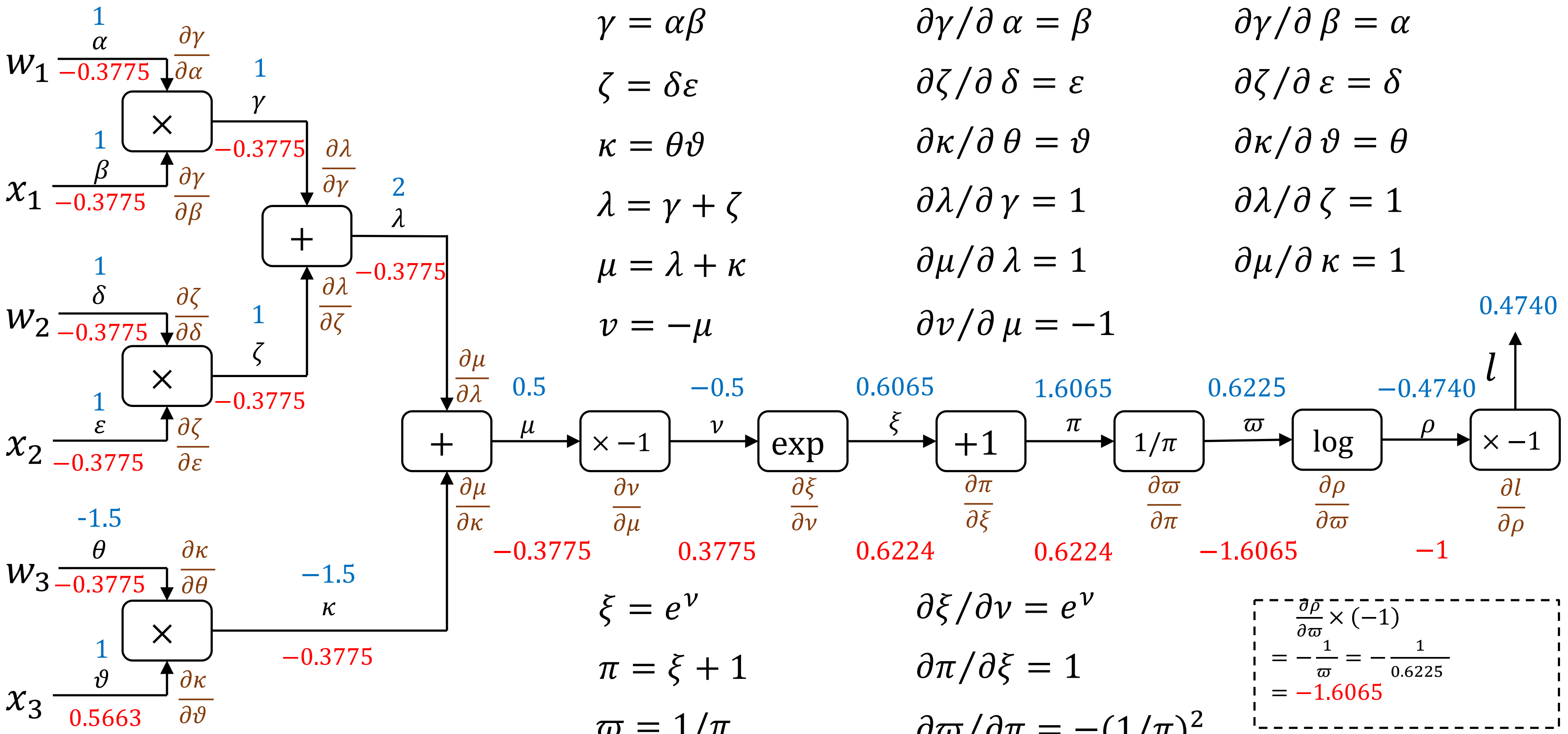
- Write a computation graph for $l_{\boldsymbol{x}}(w)$,

$$l_{\boldsymbol{x}}(w) = -\log \sigma(\boldsymbol{w} \cdot \boldsymbol{x}) = -\log \frac{1}{1 + e^{-\boldsymbol{w} \cdot \boldsymbol{x}}}$$

- Consider $\boldsymbol{x} = (1,1,1)^\top$ and $\boldsymbol{w} = (1,1,-1.5)^\top$
  - Compute the value of $l_{\boldsymbol{x}}(\boldsymbol{w})$
  - Compute gradients $\frac{\partial l_{\boldsymbol{x}}(\boldsymbol{w})}{\partial \boldsymbol{w}}$



Computing $\frac{\partial l_x(\boldsymbol{w})}{\partial \boldsymbol{w}}$ using AD



w_1 1 α −0.3775 $\frac{\partial\gamma}{\partial\alpha}$
x_1 1 β −0.3775 $\frac{\partial\gamma}{\partial\beta}$
×
1 γ −0.3775 $\frac{\partial\lambda}{\partial\gamma}$
w_2 1 δ −0.3775 $\frac{\partial\zeta}{\partial\delta}$
x_2 1 ε −0.3775 $\frac{\partial\zeta}{\partial\varepsilon}$
1 ζ −0.3775 $\frac{\partial\lambda}{\partial\zeta}$
+
2 λ −0.3775 $\frac{\partial\mu}{\partial\lambda}$
w_3 -1.5 θ −0.3775 $\frac{\partial\kappa}{\partial\theta}$
x_3 1 ϑ 0.5663 $\frac{\partial\kappa}{\partial\vartheta}$
−1.5 κ −0.3775 $\frac{\partial\mu}{\partial\kappa}$
0.5 μ −0.3775 $\frac{\partial\nu}{\partial\mu}$
× −1
−0.5 ν 0.3775 $\frac{\partial\xi}{\partial\nu}$
exp
0.6065 ξ 0.6224 $\frac{\partial\pi}{\partial\xi}$
+1
1.6065 π 0.6224 $\frac{\partial\varpi}{\partial\pi}$
1/π
0.6225 ϖ −1.6065 $\frac{\partial\rho}{\partial\varpi}$
log
−0.4740 ρ −1 $\frac{\partial l}{\partial\rho}$
× −1
0.4740 l

$\gamma = \alpha\beta$ $\quad \partial\gamma/\partial\alpha = \beta$ $\quad \partial\gamma/\partial\beta = \alpha$

$\zeta = \delta\varepsilon$ $\quad \partial\zeta/\partial\delta = \varepsilon$ $\quad \partial\zeta/\partial\varepsilon = \delta$

$\kappa = \theta\vartheta$ $\quad \partial\kappa/\partial\theta = \vartheta$ $\quad \partial\kappa/\partial\vartheta = \theta$

$\lambda = \gamma + \zeta$ $\quad \partial\lambda/\partial\gamma = 1$ $\quad \partial\lambda/\partial\zeta = 1$

$\mu = \lambda + \kappa$ $\quad \partial\mu/\partial\lambda = 1$ $\quad \partial\mu/\partial\kappa = 1$

$\nu = -\mu$ $\quad \partial\nu/\partial\mu = -1$

$\xi = e^\nu$ $\quad \partial\xi/\partial\nu = e^\nu$

$\pi = \xi + 1$ $\quad \partial\pi/\partial\xi = 1$

$\varpi = 1/\pi$ $\quad \partial\varpi/\partial\pi = -(1/\pi)^2$

$\rho = \log\varpi$ $\quad \partial\rho/\partial\varpi = 1/\varpi$

$l = -\rho$ $\quad \partial l/\partial\rho = -1$

$$\frac{\partial\rho}{\partial\varpi}\times(-1) = -\frac{1}{\varpi} = -\frac{1}{0.6225} = -1.6065$$

$$\frac{\partial\varpi}{\partial\pi}\times(-1.6065) = -\left(\frac{1}{1.6065}\right)^2\times(-1.6065) = 0.6224$$

$$\boxed{w_1 \leftarrow w_1 + \eta\frac{\partial l_x(\boldsymbol{w})}{\partial w_1} = w_1 + 0.3775\eta}$$

No need to derive backpropagation

- Manual derivation of gradients is tedious and error-prone
  - Debugging a mistake in gradients is extremely difficult

- AD is employed in most deep learning frameworks
  - We only need implement an algorithm for a forward pass, i.e., how to compute an output from an input
  - We can concentrate on designing a structure of neural network
  - This boosted the speed of research and development
  - The idea of AD is not new (since 1959)

- Deriving a formula for backpropagation is legacy

Summary and notes

- We design:
  - A neural network model $f(\boldsymbol{x}; \theta)$ (with parameters θ)
  - A loss function: $E_D(\theta) = \sum_{n=1}^{N} \mathcal{L}(f(\boldsymbol{x}_n; \theta), y_n)$
    - $\mathcal{L}$ is an instance-wise loss function
    - D presents a set of training data $D = \big((x_1, y_1), \ldots, (x_N, y_N)\big)$

- We find a minimizer θ^* for $E_D(\theta)$ by using SGD
  - An update formula for every parameter $w \in \theta$ is derived in a generic manner based on automatic differentiation

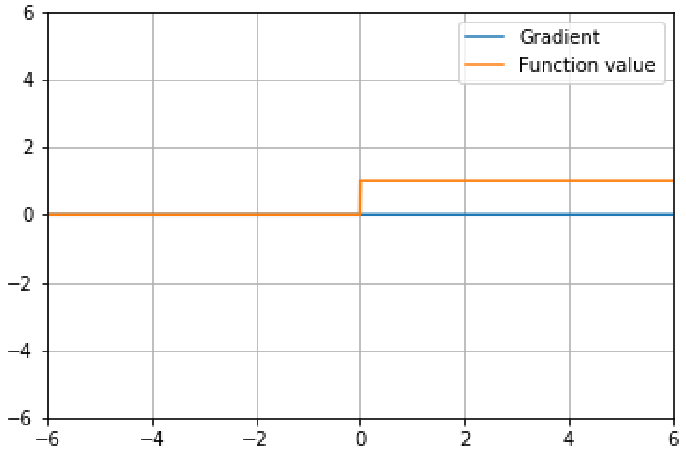
- Step function is inappropriate for backpropagation
  - Gradients will not flow because $g'(a) = 0$ at $a \neq 0$

Activation functions

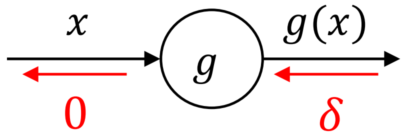
Step

Step function: $\mathbb{R} \to \{0,1\}$

$$g(x) = \begin{cases} 1 & (\text{if } x > 0) \\ 0 & (\text{otherwise}) \end{cases}$$



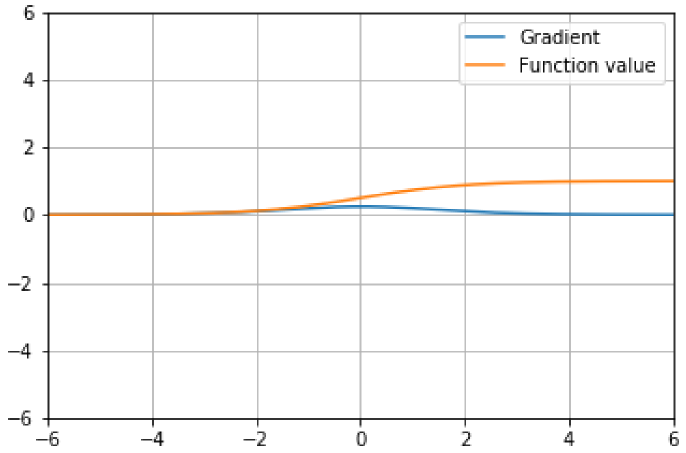
- Pros
  - Yields a binary output
- Cons (never use this)
  - Zero gradients
    - SGD cannot update parameters because $\frac{\partial l}{\partial w} = 0$



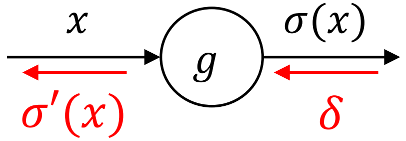
Sigmoid

Sigmoid: $\mathbb{R} \to (0,1)$

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



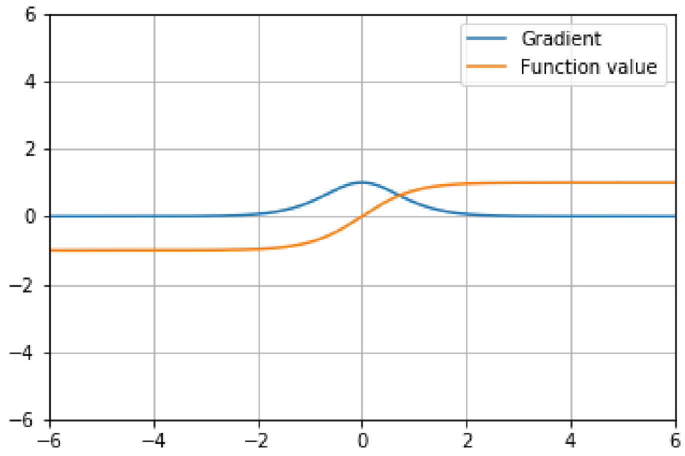
- Pros
  - Yields an output within $(0,1)$
- Cons
  - Not zero-centered
  - Zero (vanishing) gradients when $|x|$ is large



Hyperbolic tangent (tanh)

$$\tanh\colon \mathbb{R} \to (-1,1)$$

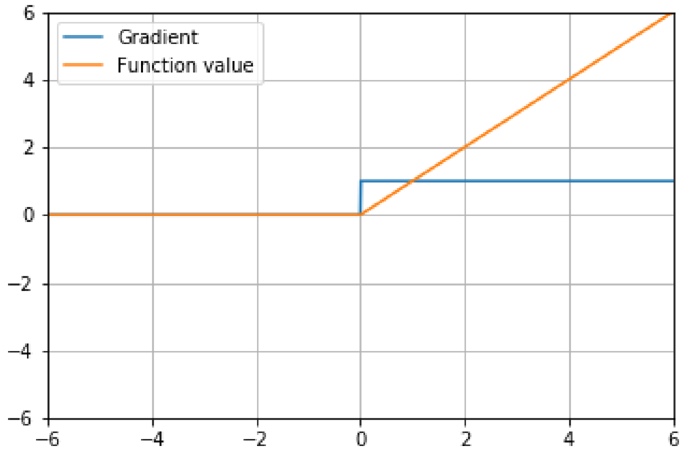
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1$$



- Pros
  - Yields an output within $(-1,1)$
  - Zero-centered
- Cons
  - Zero (vanishing) gradients when $|x|$ is large

Rectified Linear Unit (ReLU)

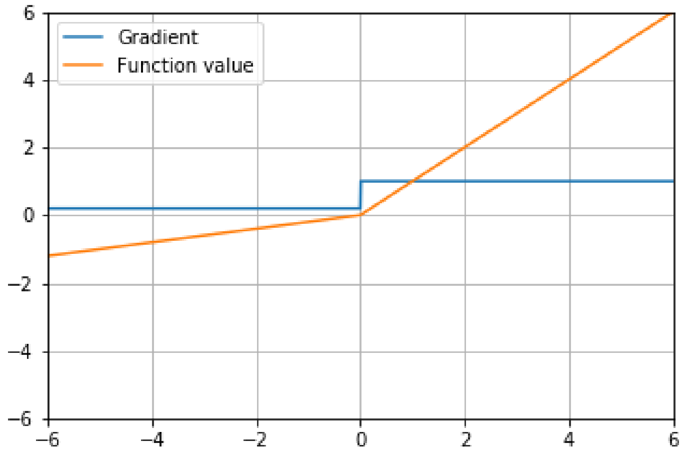
$$\text{ReLU}: \mathbb{R} \to \mathbb{R}_{\geq 0}$$
$$\text{ReLU}(x) = \max(0, x)$$



- Pros
  - Gradients do not vanish when $x > 0$
  - Light-weight (no e^x) computation
  - Faster convergence (e.g., 6x faster on CIFAR-10)
- Cons
  - Not zero centered
  - Dead neurons when $x \leq 0$

Leaky ReLU

$$\text{Leaky ReLU}: \mathbb{R} \rightarrow \mathbb{R}$$
$$\text{LeakyReLU}_{\alpha}(x) = \max(\alpha x, x)$$



- Pros
  - Gradients do not vanish
  - Light-weight (no e^x) computation
- Cons
  - Not zero centered
  - Not so much improvement over ReLU in practice

Typical definition of a DNN

Typical example of a (pseudo) definition for a Deep Neural Network of depth L (L-1 hidden layers and 1 output layer) for (almost) all modern DL frameworks and libraries:

```
initialize dnnmodel
```

dataset consists of D inputs x of dimensions d each

set hidden Layer 1 as a fully connected (fc) layer to inputs x containing n1 neurons (n1 x d connections):

```
dnnmodel.fc1(n1)
fc1.activation-function=relu
```

set hidden Layer 2 as a fully connected layer to Layer 1 containing n2 neurons (n2 x n1 connections):

```
dnnmodel.fc2(n2)
fc2.activation-function=relu
```

… more hidden layers ...

set hidden Layer L-1 as a fully connected layer to Layer L-2 containing nL neurons (nL-1 x nL-2 connections):

```
dnnmodel.fcL-1(nL-1)
fcL-1.activation-function=relu
```

set fcL (output layer) in a binary classification problem (1 x nL-1 connections)

```
dnnmodel.fcL(1)
fcL.activation-function=sigmoid
```

OR set fcL (output layer) in a K-class multiclass classification problem with one-hot encoding (K x nL-1 connections)

```
dnnmodel.fcL(K)
fcL.activation-function=softmax
```

Set Loss function to Cross Entropy (based on Maximum Likelihood Estimation). Use C for L2 weight regularization.

```
dnnmodel.loss=crossentropy(C)
```

Set the solver to mini batch SGD (or variations) with learning rate lr that by default will use automatic differentiation for the backprop gradient computation. Also set maximum number of epochs (e) and batch size (m) and use of dropout for further regularization. (one epoch equals D examples used for training)

```
dnnmodel.solver=minibatchSGD(lr,m,e,dropout=yes)
```

What made Deep Neural Networks possible and efficient?

Factors from the natural evolution of computation
- Better computers and software allow bigger networks with higher capacity to solve more difficult problems
- With bigger datasets available we must use stochastic methods like SGD

Algorithmic factors
- Cross-entropy is a better loss function than MSE for sigmoid, softmax
- ReLU in hidden layers is a better activation function than sigmoid and tanh for deeper networks
- Automatic differentiation is now a feature of all DL frameworks

What is the difference between a neural network and a deep neural network, and why do the deep ones work better?

Short answer: DNN simply seem to perform better! Read the first answer in the following link:

https://stats.stackexchange.com/questions/182734/what-is-the-difference-between-a-neural-network-and-a-deep-neural-network-and-w